Decoherence and Preferred Tensor Product Structures for Systems of Qubits

Marissa M. Singh
Pitzer College

Follow this and additional works at: https://scholarship.claremont.edu/pitzer_theses

Part of the Computer Sciences Commons, and the Quantum Physics Commons

Recommended Citation
https://scholarship.claremont.edu/pitzer_theses/186

This Open Access Senior Thesis is brought to you for free and open access by the Pitzer Student Scholarship at Scholarship @ Claremont. It has been accepted for inclusion in Pitzer Senior Theses by an authorized administrator of Scholarship @ Claremont. For more information, please contact scholarship@claremont.edu.
Decoherence and Preferred Tensor Product Structures for Systems of Qubits

Thesis by
Marissa Singh

In Partial Fulfillment of the Requirements for the
Degree of
Bachelors of Arts

KECK SCIENCE DEPARTMENT OF CLAREMONT MCKENNA COLLEGE,
SCRIPPS COLLEGE, AND Pitzer COLLEGE
Claremont, California

2023
ABSTRACT

In recent decades, the program of Decoherence has helped clarify how features of the classical world emerge from Quantum Mechanics. According to Decoherence, the interaction between a system and its environment dynamically selects certain system states — the pointer states — that exhibit predictable, classical behavior while their superpositions rapidly decohere. However, most Decoherence studies to date pre-suppose a preferred division of the world into “system” and “environment”, corresponding to a preferred choice of Tensor Product Structure (TPS) on the Hilbert Space of states. A few previous works have suggested that the existence of a well-defined pointer observable may be used to dynamically select one TPS over another (given only the barebones data of a Hilbert Space of states and Hamiltonian operator). In this thesis, we adapt the “in-principle” algorithm of [1] to a system of multiple qubits. We streamline the algorithm, incorporate the extra principle Democracy of Qubits, and code it using NumPy. Using gradient descent, the code is able to perform a search to identify an optimal TPS. Separately, the local unitary invariants of [9] are used to check whether or not the outputs of gradient descent belong to the same local unitary orbits and therefore the same TPS. We find that some TPSes are better than others for the 2-qubit system, but there does not appear to be a unique minimum TPS.
# TABLE OF CONTENTS

Abstract ......................................................... ii
Table of Contents .................................................... iii
Chapter I: Introduction ................................................. 1
Chapter II: Background ................................................. 9
  2.1 Density Matrices .............................................. 9
  2.2 Decoherence .................................................. 10
  2.3 Tensor Product Structures .................................... 14
Chapter III: Algorithm ............................................... 19
  3.1 Overview ...................................................... 19
  3.2 The Hamiltonian .............................................. 22
  3.3 Scrambling the Hamiltonian .................................. 24
  3.4 Gradient Descent ............................................. 26
  3.5 The Cost Function ............................................ 28
  3.6 Local Unitary Orbit Checks .................................. 34
  3.7 The Complete Python Program ................................ 36
Chapter IV: Methods Checks ........................................... 39
  4.1 Testing Gradient Descent ..................................... 39
  4.2 Testing Local Unitary Checks ................................. 39
  4.3 Testing CF2 .................................................. 41
Chapter V: Results and Discussion ................................. 42
  5.1 The Two Qubit Case .......................................... 42
  5.2 The Three Qubit Case ........................................ 43
Chapter VI: Conclusion and Future Directions ................. 44
Bibliography ......................................................... 46
Appendix A: Construction of Generalized Gell-Mann Matrices .... 48
Appendix B: Code Implementation ................................. 49
According to Quantum physics, small atoms seem to behave differently from macroscopic objects. In the Quantum realm, for instance, a particle can occupy two or more places at once and then instantaneously and randomly snap onto a particular location when its peace is disturbed by a position measurement. Classically, on the other hand, objects always seem to occupy one definite position, which we are able to track smoothly and predictably. However, given that the macroscopic world is built from the tiny particles that obey Quantum Mechanics, why do they seem to exhibit different behaviors? Understanding how classicality emerges from Quantum Mechanics would contribute to our understanding of Quantum Foundations and aid in the development of Quantum Computers as their function relies on the delicate interplay of Quantum and Classical Mechanics.

In the last few decades, the program of Decoherence has provided a perspective on this question. Decoherence is the process by which a system entangles with its environment, the outcome of which is to effectively keep the system snapped onto an eigenstate of one preferred observable, such as position. To explain this concept with an example, consider a composite collection of particles: a book and some photons [6]. We will designate the book to be our “system”, and the photons to be our “environment”. Every second, the photons collide with and scatter off the book. The angle of particle deflection depends on the orientation of the book, so we can therefore look to the path of the deflected photons to infer the orientation of the book. The book’s orientation is called a system observable, and the particular orientation of the book is called the system state, an eigenstate of this particular observable. Suppose the book can be in one of two distinct states, \( |\psi_1\rangle \) or \( |\psi_2\rangle \). A measurement of the environment photons allows us to infer the state of orientation without doing any violence to the state - it performs a "non-demolition" measurement characteristic of classical physics.

In Figure 1.1, the book starts out in state \( |\psi_1\rangle \) and the photons start out in ready state \( |E_0\rangle \). The ready state of the environment is a "typical" initial state. As depicted here, the ready state is a pure state, but often it is chosen to be a mixed, thermal state at a specified temperature. After the time evolution in which they interact, the photons
Figure 1.1: Case 1: In this case, we start the book off in a pointer state \( |2 \rangle \). By doing so, our book retains coherence and is robust to entanglement with the environment. Measuring \( |E_1 \rangle \) does not disturb the state of the book.

If the book had instead been in state \( |\psi_2 \rangle \), the photons would have evolved into state \( |E_2 \rangle \). In this case, the book starts out in an unentangled state with the environment and ends in an unentangled state. To be unentangled means that a composite system can be written as a tensor product of the state vectors of its sub-parts (i.e. \( |\psi_{sys} \rangle \otimes |\psi_{env} \rangle \)). Because the system remains unentangled, we say that the book alone has "maintained coherence", as a subsequent measurement of the photons would give us information about the state of the book without disturbing the state. Because there is no loss of predictability, we can acknowledge that Case 1 represents the emergence of classical behavior.

Figure 1.2: Case 2: In this case, we start the book off in a superposition of orientation states \([2\rangle\). As a result of the book becoming entangled with the environment after the time evolution, we say that the book alone has "lost coherence". Measuring the photons would randomly snap the book onto an orientation state, returning it to Case 1.

If we instead started our book out in a superposition of states, the composite state after the time evolution does not remain in an unentangled state. In Figure 1.2, the book starts out in a superposition of states \( |\psi_1 \rangle \) and \( |\psi_2 \rangle \), and the environment starts
out in state $|E_0\rangle$. The superposition of states that was seen in the book spreads to the composite system, and the initial coherence of the system is now a shared property of the composite system-environment state. Because of this shared coherence, in order to do a predictable measurement, one would need to perform a joint measurement of all the photons and the book. In practice, the environment usually has many degrees of freedom (as opposed to the lesser degrees of freedom of the few photons we are considering in this example). It would be impractical to measure the precise state of every particle in the universe. Effectively, an observer embedded in the environment would say that book has entered an incoherent "mixed" state: no matter what system observable is measured, at the final moment, results will be uncertain. Further, a measurement of either system or environment alone disturbs the state of the book. From the point of view of an observer-embedded environment, Case 2 snaps back onto Case 1.

In fact, the reason Case 1 did not decohere was because the system started out in one of a special class of states called pointer states. Coined by Wojciech Zurek, one of the main developers of Decoherence, pointer states are quantum versions of the possible states we typically observe in the classical world. They are the eigenstates of the pointer observable, which in this case is the orientation of the book. As Zurek discusses, pointer states are identified as the ones that behave most classically because they are predictable. Zurek refers to the process of finding these states as the predictability sieve [12]. As exemplified in Case 1, pointer states are resistant to entanglement with the environment, and hence they are resistant to decoherence. Although Case 2 was not resistant to decoherence, it involved starting the system out in a superposition of pointer states, allowing for the possibility of a return to Case 1. For either Case 1 or Case 2 to exist, there must be the existence of a well-defined pointer observable.

As it is relevant to this thesis, we mention a further possibility. Suppose that a certain system had no well-defined pointer observable, such as orientation. In this case, the system would be continually entangled with the environment. No system observable could be predictably tacked. We would have to conclude that the system resists any emergent classical description. The Decoherence explanation of classicality depends on systems such as books admitting a well-defined pointer observable. This Case is illustrated in Figure 1.3.

One special quality of Decoherence is that when a system decoheres, there is an increase in entanglement entropy due to the resulting system-environment entangle-
Figure 1.3: Case 3: In this case, there is no well-defined pointer observable. The system continually entangles with its environment offering no possibility to return to Case 1.

The increase of entanglement entropy allows us to measure whether or not a state has decohered. Entanglement entropy will be further detailed in Subsection 2.2 through an explicit example.

The foregoing is part of the standard account of Decoherence. However, there is an observation to be made in declaring Decoherence to be an effective tool for exploring the emergence of classicality from Quantum Mechanics. Notice that the account of Decoherence described above pre-supposes a certain division into "system" and "environment". The picture of the book and photons we drew in Figures 1.1-1.2 was already from the point of view of a classical observer, informed by how these objects familiarly look and behave: the "book" part was drawn using a different graphic from the "photons" part [6]. But, the universe holds no bias toward how a composite system is divided into its sub-parts. The ingredients of barebones Quantum Mechanics presented in textbooks are just a Hilbert Space of states $\mathcal{H}$ along with a particular hermitian operator $\hat{H}$ singled out as the Hamiltonian generating time evolution. Hence, to explain the emergence of classicality from quantum mechanics means to identify classical behavior given only $\mathcal{H}$ and $\hat{H}$. A particular division into sub-systems is an extra layer of structure and it is unsatisfactory to "derive" classicality by assuming this.

"In particular, one issue which has been often taken for granted is looming big, as a foundation of the whole decoherence program. It is the question of what are the 'systems' which play such a crucial role in all the discussions of the emergent classicality." - Wojciech Zurek [13]
To better exemplify this change in perspective, consider a composite collection of two qubits. Qubits are 2-dimensional systems; for example, spin-1/2 particles. The quantum description of the joint system is as follows:

The total Hilbert Space is the tensor product of the two individual Hilbert Spaces:

\[ \mathcal{H}_{\text{total}} \approx \mathcal{H}_1 \otimes \mathcal{H}_2. \]  

(1.1)

The total Hamiltonian can be uniquely decomposed into the sum of both self-Hamiltonians combined with the interaction Hamiltonian that couples the two qubits:

\[ \hat{H}_{\text{total}} = \hat{H}_1 \otimes \hat{I} + \hat{I} \otimes \hat{H}_2 + \hat{H}_{\text{int}}. \]  

(1.2)

In writing Equation 1.2, we have built up a composite from sub-parts, thus inducing a certain splitting of \( \mathcal{H} \) in the process. Akin to working with two slices of pizza, it is easy for us to put two halves together to form a whole.

![Figure 1.4: Tensor Product](image)

Figure 1.4: Tensor Product: Similar to how there is a unique way to combine two halves of a pizza to form a whole pizza, it is just as straightforward when tensoring the Hilbert Spaces of two qubits to compute the composite Hilbert Space.

Starting with \( \mathcal{H}_1, \mathcal{H}_2 \) and then tensoring automatically induces a certain division into "system" and "environment". What is less explored is going in the reverse direction: expressing \( \mathcal{H} \) as in Equation 1.1. If we instead assumed Barebones Quantum Mechanics, meaning we only have access to the total Hilbert Space and the total Hamiltonian, it is no longer as clear how one should divide the composite system into sub-parts. As illustrated in Figure 1.5, we now have the freedom to cut the pizza in many different ways. By analogy, to select one or a sub-class of splitting requires a selection criterion - a "Cost Function" if you will, among the various options.

This process of dividing complete system \( \mathcal{H} \) into sub-parts is called tensor product factorization [1] [3] [11]. A specific division into sub-parts is called a Tensor
\[ \mathcal{H} \cong \mathcal{H}_1 \otimes \mathcal{H}_2 \]

Figure 1.5: Tensor Product Factorization: Going the reverse direction of dividing a pizza into sub-parts given only a complete pizza, it becomes unclear how we should divide the pizza. As demonstrated, we now have the freedom to cut the pizza in any way we would like. This same freedom applies to factorizing a Tensor Product Structure given only the total Hamiltonian and total Hilbert Space of some system.

Thus far, it is clear that the emergence of classicality relates to the existence of states robust to entanglement. As seen in our book example, the way we divided up the composite system allowed for Case 1 to exist, meaning that due to the particular
division we chose, there existed a well-defined pointer observable. This leads us to
the question: are some divisions better than others in the sense of admitting some
pointer observable? Or more formally, are all Tensor Product Structures equally
good for giving an account of Decoherence? If not, is there a unique TPS or a
discrete collection of TPSes that allows for Decoherence?

Building on the work of those who have previously tried to answer these questions,
this thesis seeks to create a quantitative criterion for selecting a preferred Tensor
Product Structure in which features of the classical world emerge. Using barebones
Quantum Mechanics, Max Tegmark, Ashmeet Singh, and Sean Carroll explored
a criterion of robustness for selecting a Tensor Product Structure [1] [10]. Their
conclusion was that the existence of a well-defined pointer observable might select
a preferred Tensor Product Structure. However, because their system and selection
criteria were complex, they called their criteria an "in principle" algorithm in the
sense that it could not be implemented on a computer [1]. Jordan Cotler, et al. also
explored a similar question but instead used locality of the Hamiltonian as a criterion
[3]. They found that there are typically only one or a few Tensor Product Structures
in which the Hamiltonian has a local structure. Aspects of these questions have also
been addressed in the work of Zanardi, Kabernik, and Pollack [11] [4]. The goal of
this thesis is to optimize previous methods to decrease computational expense and to
be able to check whether our quantitative criterion is able to select a preferred Tensor
Product Structure. Additionally, our method allows us to go beyond the "Quantum
Measurement Limit" (of [1]) and to compute a score for an entire TPS efficiently
as an eigenvalue of a certain matrix. Finally, we re-purpose the local unitary (LU)
invariants of [9] to detect whether or not two bases correspond to the same TPS. We
did this by exploring simple systems of qubits, and creating a simplified version of
a selection criterion that combines the considerations of Tegmark, Singh, Carroll,
and Cotler, et al., while also including a new criterion of our own.

In this thesis, I detail the process I took to construct a function runnable in Python
that (1) takes in a choice of a number of qubits (equivalent to a choice of $\mathcal{H}$), (2) takes
in a choice of Hamiltonian $\hat{H}$, (3) scrambles the qubits so that they are representative
of a particular Tensor Product Structure, (4) evolves the scrambled system forward
in time, (5) takes the partial trace, and (6) assigns a score to that tensor product
factorization based on the measured increase of entanglement entropy. Thus, the
score of a particular TPS corresponds to the optimal score of the observable. We
are exploring systems of 2-3 qubits that are governed by one of four different
Hamiltonians: The Spin Bath Model, the 1D Ising Model, the C-NOT Model, and the 3D Heisenberg Model. With this function as a basis, a separate Python program starts off at a completely randomized Tensor Product Structure, computes the score of that structure, and then uses a gradient descent algorithm to look through the space of Tensor Product Structures for the preferred Tensor Product Structure with the minimum entanglement entropy score of 0. With additional modules, the newfound Tensor Product Structure is checked to see if it is the same Tensor Product Structure that we originally started with – the structure in which the composite system is divisible by qubits, and our classical world emerges.

This paper is organized as follows: Subsection 2.1 describes the formalism of density matrices. Subsection 2.2 gives a more detailed account of what Decoherence is, including a quantitative example in which we can see differences in entanglement entropy. Subsection 2.3 discusses Tensor Product Structures in depth, explaining how one can abstractly visualize a space of Tensor Product Structures, and how one utilizes unitary transformations to move across different Tensor Product Structures. Section 3 lays out details of the algorithm with each subsection breaking down the motivation behind using particular functions and methods. Section 4 discusses the methods used to test the gradient descent algorithm, as well as the module used to check whether or not two bases are from the same tensor product structure. Section 5 discusses the results found thus far, and Section 6 includes the conclusion and directions for future research.
Chapter 2

BACKGROUND

In this chapter, we make the description of the preceding chapter quantitative and illustrate the computational tools used in later sections by stepping through concrete examples in detail.

2.1 Density Matrices

In the study of Decoherence, it is useful to represent states of a system not with kets, but with density matrices. When a system (\(\psi_{sys}\)) becomes entangled with its environment (\(\psi_{env}\)), the state of the composite system cannot be expressed in tensor product form (\(|\psi_{sys}\rangle \otimes |\psi_{sys}\rangle\)), and we are not able to consider the system and environment as separate entities. The system alone cannot be assigned a pure state but rather, a mixed state (i.e. a statistical mixture of pure states). Thus, it is impossible to assign an individual quantum state vector to the system. Density matrices represent both pure and mixed states equally well, encoding the statistics of every possible measurement made on the system alone [6].

To put a pure state in density matrix form, we compute:

\[
\hat{\rho} = |\psi\rangle \langle \psi|,
\]  

(2.1)

by choosing a particular basis and assorting the results into a matrix. For instance, consider a composite system of two qubits where qubit 1 is our “system”, and qubit 2 is our “environment”. Qubits are 2-state systems, such as spin-1/2 particles. They have an orthogonal basis with the following two states: spin up, corresponding to the \(|0\rangle\) qubit state, or spin down, corresponding to the \(|1\rangle\) qubit state. If both qubits are in the \(|0\rangle\) state, the composite state can be expressed as:

\[
|\psi\rangle = |0_1\rangle|0_2\rangle,
\]  

(2.2)

with corresponding density operator:

\[
\hat{\rho} = |0_1\rangle|0_2\rangle\langle 0_1|\langle 0_2|.
\]  

(2.3)

We can assort these results into a four-by-four matrix in the standard way, according to the table in Figure 2.1.
Figure 2.1: Matrix Guide: This is a general guide for organizing a density operator into a matrix for a system of two qubits. The same methodology can be generalized to any tensor product space.

Thus our matrix would be:

$$\rho = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$  \hfill (2.4)

Suppose the environment is inaccessible or contains too many degrees of freedom to measure in practice. Effectively, we only have access to the outcomes of measurements of observables of the form $\hat{A}_{sys} \otimes \hat{I}_{env}$. Assuming that we only have access to the system, we can reduce the full density matrix so that $\rho$ to a density matrix $\rho_{sys}$ encoding statistics of our restricted observables. To do this, we take a partial trace over the environment of our full density matrix $\hat{\rho}$:

$$\rho_{sys} = Tr_{env}(\hat{\rho}) = \langle 0_2 | \hat{\rho} | 0_2 \rangle + \langle 1_2 | \hat{\rho} | 1_2 \rangle.$$  \hfill (2.5)

Computing the reduced density matrix for our example, we get:

$$\rho_{sys} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$  \hfill (2.6)

The measurement statistics for all of the observables only pertaining to the system alone are completely encoded in $\rho_{sys}$. This proves to be a very useful tool in the formal description of Decoherence, as we will see in the following subsection.

### 2.2 Decoherence

Recall from Chapter 1 that Decoherence helps explain the emergence of classicality from Quantum Mechanics. Decoherence is the process by which a system entangles
with its environment, effectively “measuring” the state of the system. Decoherence was first introduced by Heinz-Dieter Zeh in 1970 and further developed by Wojciech Zurek and collaborators in subsequent decades [5] [12] [13].

Recall that entanglement entropy can be used to measure the degree to which a state has decohered. To continue with the example above, consider a composite system of two qubits. Akin to the book example, we will start our system out in the initial state $|0\rangle_1$, and have our environment start in a ready state. Usually in models of Decoherence, the ready state of the environment is a "typical" thermal state $\hat{\rho}_\text{ready} = \frac{e^{-\frac{\hat{H}}{kT}}}{\text{Tr}(e^{-\frac{\hat{H}}{kT}})}$. Our choice represents the infinite temperature limit and is motivated in Subsection 3.1 as being technically convenient [1]. With $d_{\text{qubit}} = d_{\text{sys}} = d_{\text{env}} = 2$, our composite initial state is:

$$\rho_0 = \rho_{0,\text{sys}} \otimes \rho_{0,\text{env}} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes \frac{1}{d_{\text{env}}} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$ (2.7)

In the book example, we saw when a system started out in one of its pointer states, after time evolution, the composite system remained unentangled. In this example,
we will be evolving our initial state forward in time according to one of the models we use later in this paper: the 1D Ising Model. As we will show, the initial state we chose turns out to be a pointer state for this particular $\hat{H}$, demonstrating Case 1. We are defining $\hat{H}_{Ising}$ by writing it in a "native" basis where it takes a simple form:

$$H_{Ising} = \begin{pmatrix}
-1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & -1
\end{pmatrix}.$$  \hspace{1cm} (2.8)

Following Carroll and Singh, the specific time the initial states are set to evolve to is called the characteristic time $[1]$, defined as:

$$t_{\text{char}} = \frac{1}{\|\hat{H}\|_2} = 1,$$  \hspace{1cm} (2.9)

where $\|\hat{H}\|_2$ is a unitarily invariant matrix norm \(^1\). After evolving the initial state forward to the characteristic time according to the equation:

$$\hat{\rho}_f = e^{-it\hat{H}} \cdot \hat{\rho}_0 \cdot e^{it\hat{H}},$$  \hspace{1cm} (2.10)

the state of the composite system becomes:

$$\rho_f = \begin{pmatrix}
0.5 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.5 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0
\end{pmatrix}.$$  \hspace{1cm} (2.11)

Due to the fact that $[\hat{H}, \hat{\rho}_0] = 0$, the initial state is equal to the final state. Assuming that we only have access to the system, we compute the reduced density matrix:

$$\rho_{f,\text{sys}} = \begin{pmatrix}
1.0 & 0.0 \\
0.0 & 0.0
\end{pmatrix}.$$  \hspace{1cm} (2.12)

Our final step is to examine the change in entanglement entropy. We do this by computing the purity entropy:

$$S = 1 - Tr\left(\hat{\rho}_{\text{sys}}^2\right),$$  \hspace{1cm} (2.13)

\(^1\)Advantage of $\|\hat{H}\|_2 = \|\hat{U} \cdot \hat{H} \cdot \hat{U}^\dagger\|_2$ is that we will later be acting with unitary operators and wish to have a common benchmark at which to BLANK out $S$.  

which quantifies how mixed the state $\hat{\rho}_{sys}$ is; equivalently, how entangled the system is with the environment. Recall that our initial state of the system is just $\hat{\rho}_{0,sys}$. Computing the purity entropy of $\hat{\rho}_{0,sys}$ and $\hat{\rho}_{f,sys}$, we get:

\[ S_{\text{initial}} = 1 - Tr \left( \rho_{0,sys}^2 \right) = 1 - Tr \left( \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \right) = 1 - 1 = 0. \quad (2.14) \]

\[ S_{\text{final}} = 1 - Tr \left( \rho_{f,sys}^2 \right) = 1 - Tr \left( \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \right) = 1 - 1 = 0. \quad (2.15) \]

Because the before and after states ended up being exactly the same, we can see that there was no increase in entanglement entropy after the time evolution and that the entropy remained at 0. We have learned that $|0_1\rangle$ with density matrix $\hat{\rho}_{0,sys} = |0_1\rangle\langle 0_1|$, is a pointer state. Note that the choice of the unentangled initial state and partial trace procedure $Tr_{\text{env}}$ are already adapted to the particular system/environment split. A pointer observable is a pair of orthogonal states; in this case $|0_1\rangle\langle 0_1|$ and $|1_1\rangle\langle 1_1|$. These states are always antipodal points on the Bloch Sphere, illustrated in Figure 2.2. In Subsection 3.5, we will perform an explicit calculation and show that $S_{\text{final}}$ of each is always the same. Hence, $|1_1\rangle\langle 1_1|$ is also a pointer state.

We choose a different starting state: a superposition of pointer states $\frac{1}{\sqrt{2}}(|0_1\rangle + |1_1\rangle)$ with density matrix $\hat{\rho}_0 = \frac{1}{\sqrt{2}}(|0_1\rangle\langle 0_1| + |0_1\rangle\langle 1_1| + |1_1\rangle\langle 0_1| + |1_1\rangle\langle 1_1|)$. The initial state of the composite system becomes:

\[ \rho_0 = \rho_{0,sys} \otimes \hat{\rho}_{0,env} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \otimes \frac{1}{d_{\text{env}}} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{4} & 0 & \frac{1}{4} & 0 \\ 0 & \frac{1}{4} & 0 & \frac{1}{4} \\ \frac{1}{4} & 0 & \frac{1}{4} & 0 \\ 0 & \frac{1}{4} & 0 & \frac{1}{4} \end{bmatrix}. \quad (2.16) \]

Evolving the initial state forward to the characteristic time, the state of the composite system becomes:

\[ \rho_f = \begin{bmatrix} 0.250 & 0.0 & -0.104 + 0.227 i & 0.0 \\ 0.0 & 0.250 & 0.0 & -0.104 - 0.227 i \\ -0.104 - 0.227 i & 0.0 & 0.250 & 0.0 \\ 0.0 & -0.104 + 0.227 i & 0.0 & 0.250 \end{bmatrix}. \quad (2.17) \]

Reducing the density matrix to be representative of only the system:

\[ \rho_{f,sys} = \begin{bmatrix} 0.500 & -0.208 \\ -0.208 & 0.500 \end{bmatrix}. \quad (2.18) \]
Examining the change in entanglement entropy between $\hat{\rho}_{0,\text{sys}}$ and $\hat{\rho}_{f,\text{sys}}$:

\[ S_{\text{initial}} = 1 - \text{Tr}\left(\hat{\rho}_{0,\text{sys}}^2\right) = 0. \]  \hfill (2.19)

\[ S_{\text{final}} = 1 - \text{Tr}\left(\hat{\rho}_{f,\text{sys}}^2\right) = 0.413. \]  \hfill (2.20)

In line with the book example, we can see an increase in entanglement entropy due to starting our system out in a superposition of its pointer states. This feature of decoherence will prove to be effective in determining whether or not our composite system decohered.

### 2.3 Tensor Product Structures

An assumption of textbook Quantum Mechanics is that the Hilbert Space of States $\mathcal{H}$ of a composite system is the tensor product of individual Hilbert Spaces $\mathcal{H}_1, \mathcal{H}_2,...$ of its subsystems:

\[ \mathcal{H} \approx \mathcal{H}_1 \otimes \mathcal{H}_2 \otimes ... \otimes \mathcal{H}_n, \]  \hfill (2.21)

where $\approx$ denotes the isomorphism of vector spaces. Consider now the reverse operation: starting with $\mathcal{H}$ of a composite system, and decomposing $\mathcal{H}$ with different isomorphisms $\approx$ in Equation 2.21, corresponding to different divisions into "system" and "environment". A tensor product structure (TPS) formalizes the particular division of a composite system into its sub-parts. Using the analogy of slicing pizzas in the previous Section, we noted that the same $\mathcal{H}$ can admit many distinct TPSes. We wish now to precisely define a TPS.

We write down the definition below, but let us motivate it. It is useful to think of a TPS as corresponding to an equivalence class of labelings of basis directions in $\mathcal{H}$. Going back to the qubit example, we were working with the orthogonal z-basis states $\{|0\rangle, |1\rangle\}$ for both the system and the environment. If we instead partitioned the composite system in a different way, we would have different basis labelings.

To demonstrate this idea, we can visualize the 4-dimensional space of this tensor product structure using 3 directions of which we illustrate in Figure 2.3. In the graph, each axis corresponds to a possible state of the composite system. For a joint system of 2 qubits, there are four possible states: $|01\rangle|02\rangle$, $|01\rangle|12\rangle$, $|11\rangle|02\rangle$, $|11\rangle|12\rangle$. As seen in Figure 2.3, the axes are labeled with these states, excluding the $|11\rangle|11\rangle$ state as we are not able to visualize four dimensions.

Suppose the red vector represents the post-evolution state. According to the labeling at the top of the figure, this is an entangled state of the two qubits (i.e. the initial
Figure 2.3: Tensor Product Structure Visualization: We can visualize the four-dimensional space of a TPS for a two-qubit-system, ignoring the $|1_1⟩|1_2⟩$ state of the system. The green vector is a post-evolution state. As shown here, it is an entangled state.

state that led to this was not a pointer state). On the other hand, as seen in Figure 2.4, this same state in $\mathcal{H}$ would be unentangled relative to a different basis labeling.

Figure 2.4: Unitary Change of Basis: Note that the post-evolution state (green), which was entangled in the basis of Figure 2.3, is now an unentangled state with respect to this new labeling. We say that the rotation pictured here has "changed TPS" because the split between particles 1 and 2 has been fundamentally altered. This is analogous to rotations of the pizza.

However, note that not all changes of basis alter the split between "system" and
"environment". Suppose that $|0_{1}\rangle' = \frac{1}{\sqrt{2}}(|0_{1}\rangle + |1_{1}\rangle)$ and $|1_{1}\rangle' = \frac{1}{\sqrt{2}}(|0_{1}\rangle - |1_{1}\rangle)$. The basis $|0_{1}\rangle' \otimes |0_{2}\rangle$, $|0_{1}\rangle' \otimes |1_{2}\rangle$, ... preserves which states are entangled or unentangled.

Now, for the formal definition, see [3] for an especially clear account. A TPS $T$ of a Hilbert Space $\mathcal{H} \approx C^{n}$ is an equivalence class of isomorphisms

$$T : \mathcal{H} \to \mathbb{C}^{d_{\text{sys}}} \otimes \mathbb{C}^{d_{\text{env}}} \quad (2.22)$$

modulo change of basis for $\mathcal{H}_{1}, \mathcal{H}_{2}$ individually and permutations of factors (which re-label which factor is "1" and which is "2") [3]. This can be thought of as the set of $\hat{U}(d_{\text{sys}}d_{\text{env}})$ matrices (each one corresponding to a different basis labeling) modulo equivalence under local unitaries (of the form $\hat{U}_{1}(d_{\text{sys}}) \times \hat{U}_{2}(d_{\text{env}})$) and permutation of factors. This formalizes the process of splitting $\mathcal{H}$ into a $d_{\text{sys}}$ and a $d_{\text{env}}$ sub-system.

A TPS is a certain equivalence class of $SU(D)$ matrices $[U(\theta)]$ (as $\theta$ ranges over all values).

For the purposes of finding a TPS in which classicality emerges, we wish to assign a score $S$ to each TPS, thought of as an equivalence class of unitaries. Hence if $[\hat{U}] = [\hat{U}']$ by virtue of $\hat{U}' = \hat{U} \cdot (\hat{U}_{1} \otimes \hat{U}_{2})$, then we should have $S(\hat{U}) = S(\hat{U}')$. As emphasized in [3], it is useful to shift perspective to think of mapping $\hat{H}$ to a different $\hat{H}' = \hat{U} \cdot \hat{H} \cdot \hat{U}^{\dagger}$ while leaving basis labelings the same. Thus, we can adopt the dual viewpoint (as in [1] [3]):

A TPS is a certain equivalence class of Hermitian $D \times D$ matrices $H$, related to $H_{\text{native}}$ by unitary transformations.

As we move between different $SU(D)$ elements, we will move from one TPS to another. To understand how a unitary transformation affects a Tensor Product Structure, we can first visualize the space of Tensor Product Structures as a collection of lines. In the visualization illustrated in Figure 2.5, each line represents a local unitary (LU) orbit. A discrete collection of lines related by permutation matrices is a TPS. Within a particular Tensor Product Structure, there are many ways of defining the basis of the system and the basis of the environment. Sliding up and down a TPS line represents a combination of particular basis labelings that do not alter the division between system and environment. Moving across lines alters the
cut between tensor product sub-factors. We choose one particular basis in which to write the Hamiltonian defined as $\hat{H}_{\text{native}}$. Our choice is one of the inputs of the toy model (the toy models that we chose to explore are further detailed in Subsection 3.2), chosen so that the form of $\hat{H}_{\text{native}}$ is simple and has a well-defined pointer observable.

![Figure 2.5: Unitary Orbit $O(\hat{H})$ of $\hat{H}_{\text{native}}$: Each point in this figure represents some hermitian matrix $\hat{H}''$ related to $\hat{H}_{\text{native}}$ by a unitary transformation: $\hat{H}'' = \hat{U} \cdot \hat{H}_{\text{native}} \cdot \hat{U}^\dagger$. Each line represents an LU orbit $O_{LU}(\hat{H}) \subset O(\hat{H}) : \hat{H}' = (\hat{U}_1 \otimes \hat{U}_2) \cdot \hat{H} \cdot (\hat{U}_1 \otimes \hat{U}_2)^\dagger$. Discrete collection of lines is a TPS (the red lines in the Figure): $\hat{H}''' = \Pi \cdot \hat{H}_{\text{native}} \cdot \Pi$.]

Such a transformation takes the form:

$$O(\hat{H}) = \{ \hat{U} \cdot \hat{H} \cdot \hat{U}^\dagger \text{ for all } \hat{U} \},$$

(2.23)

where $\hat{U} = \hat{U}(\theta)$ are $2d_2 \times 2d_2$ special unitary matrices. The special unitary matrices are parameterized by a list of variables defined as $\theta = \{ \theta_1, \theta_2 ... \}$. These thetas are applied to a set of traceless Hermitian generators which are then summed, multiplied by $i$, and exponentiated to give us the special unitary matrix $\hat{U}(\theta)$. The process of parameterizing the unitary matrices $\hat{U}$ acting on $\hat{H}$ using a list of $\theta$ parameters will be further detailed in Subsection 3.3.

2We can assume without loss of generality that $\hat{U}$ lie in $SU(2d_2)$, with determinant one, since the overall phase of the matrix has no effect when it acts on $\hat{H}$. 
To move up and down a TPS line, also known as a local unitary orbit, one can perform unitary transformations that take the form:

\[ O_{LU}(\hat{H}) = \left\{ (\hat{U}_1 \otimes \hat{U}_2) \cdot \hat{H} \cdot (\hat{U}_1^\dagger \otimes \hat{U}_2^\dagger), \quad \text{for all } \hat{U}_1, \hat{U}_2 \right\}, \tag{2.24} \]

where \( \hat{U}_1 \) are \( 2 \times 2 \) special unitary matrices and \( \hat{U}_2 \) are \( d_2 \times d_2 \) special unitary matrices. For this case, we must parametrize the unitary matrix using a subset of the list of thetas mentioned above. This subset will be further detailed in Subsection 3.3.

We can also move through the space of unitary orbits through permutation matrices. Permutation matrices relegate the role of "system" to another sub-part of the composite system. When applying a permutation matrix, we do not alter the division between "system" and "environment", thus lines related by permutation matrices are a part of the same TPS.

Our goal is to assign a score to each TPS where a TPS corresponds to a set of lines related by permutations in Figure 2.5. Using methods formulated by Sun et al.[9], we are able to check whether or not two matrices belong to the same TPS. This process is further detailed in Subsection 3.6.
Chapter 3

ALGORITHM

As described in previous Sections, we suppose that we are handed the data of what we have termed barebones Quantum Mechanics: a Hilbert Space of states $\mathcal{H}$ (which corresponds to a choice of size since all n-dimensional complex vector spaces are isomorphic) along with a particular traceless, hermitian operator $\hat{H}$ (written in some basis $\hat{H}_{n_{a_{r}{v}{i}t}{e}}$. (Also, implicitly, we have the extra notion that we will be dividing $\mathcal{H}$ into two-dimensional sub-factors; for future research, one would eventually aim to relax this assumption.) Using only these ingredients, our task is to develop a procedure that assigns a score of $S$ to each TPS (i.e. each equivalence class $[\hat{H}]$ of matrices $\hat{H}$) in order to test whether some are better than others and whether there may be a unique optimal one for allowing emergent classicality. Our scoring algorithm is motivated by Carroll and Singh’s research in which they proposed that the existence of a well-defined pointer observable can be a selection criterion for selecting a preferred Tensor Product Structure, the python program detailed below is a modified version of their Quantum Mereology algorithm [1].

3.1 Overview

![Flow Chart](image)

Figure 3.1: Overview: This is a flow chart depicting the general process of the algorithm. We will be referencing where we are in this flow chart in the following Subsections.

To start with a brief overview, the program begins by taking in a specified number of qubits ($n$) and a Hamiltonian ($\hat{H}_{n_{a_{r}{v}{i}t}{e}}$) chosen to have a well-defined pointer observable. Because we are consistently choosing qubit 1 to be our “system” and
the rest of the qubits to be the “environment”\textsuperscript{1}, our dimensions are:

\begin{align*}
    d_{\text{sys}} &= 2, \\
    d_{\text{env}} &= 2^n, \\
    D_H &= d_{\text{sys}}d_{\text{env}}.
\end{align*}

Once the Hamiltonian is constructed, the algorithm scrambles it into a random Tensor Product Structure, as illustrated in Figure 3.2. The Hamiltonian is scrambled by means of a unitary transformation:

\[ \hat{H} \rightarrow \hat{H}' = \hat{U}(\theta) \hat{H} \hat{U}(\theta)^\dagger. \]

For the reasons expressed in Subsection 2.2, the system starts out in a Candidate Pointer State and the environment starts out in a maximally-mixed ready state. Since

\textsuperscript{1}Although owing to our Democracy of Qubits criteria described below, each qubit will take turns as the "system"
\( \hat{U}_1 \) acts by rotating the Bloch Sphere for qubit 1, as we act with different \( \hat{U}_1 \) (moving up and down a LU orbit line), the \( |0\rangle \langle 0| \) state will eventually test out every possible Candidate Pointer State \( |n\rangle \langle n| \) of the system. This gives us the initial states:

\[
|0\rangle \langle 0| \otimes \frac{1}{d_{env}} \hat{I}, \quad (3.5)
\]

\[
|1\rangle \langle 1| \otimes \frac{1}{d_{env}} \hat{I}, \quad (3.6)
\]

which together define a Candidate Pointer Observable.

Figure 3.3: Initial State for System of Three Qubits: We can visualize each qubit using the Bloch Sphere. Qubit 1’s initial state can be described by any \(|+n\rangle \) or \(|-n\rangle \) vector. We choose the initial state of qubit 1, representative of the "system", to be the \(|0\rangle \) and \(|1\rangle \) Candidate Pointer States. Qubits 2 and 3, representative of the "environment" are in a maximally mixed ready state depicted by the black dot at the center of the Bloch Sphere.

The initial states and the scrambled Hamiltonian (\( \hat{H}_{scram} \)) are then fed to the gradient descent module. Recall from Subsection 2.3 that we wish to assign a single score to an entire TPS. Thus, each score represents the best score over an entire TPS. Because the system is now being evolved according to the scrambled Hamiltonian, there are different pointer states that correspond to \( \hat{H}_{scram} \), and we will see non-zero entanglement entropy growth after a time evolution. The gradient descent algorithm works to “unscramble” the matrix by repeatedly constructing new specified unitary matrices that are applied to \( \hat{H}_{scram} \) to minimize the entropy growth. Once the algorithm finds the Hamiltonian that produces 0 entropy growth, the Hamiltonian is put through a series of checks to see whether or not it lays in the same local unitary orbit as \( \hat{H}_{native} \). If they share the same local unitary orbit, we can conclude that they belong to the same Tensor Product Structure. If they are different, we can confirm
that there are Tensor Product Structures outside of the native one in which classical behavior emerges.

### 3.2 The Hamiltonian

There are four choices of Hamiltonians for \( n \)-qubit systems that we are currently exploring: The Spin Bath Model, the 1D Ising Model, the C-NOT Model, and the 3D Heisenberg Model. The reason we chose to explore these models is that they are most commonly useful in other areas of research and in toy models of Decoherence [6]. The first two models hold slightly more importance as they highlight certain characteristics for systems where \( n > 2 \).

In the Spin Bath Model, there is a central qubit to which each qubit is coupled. In Figure 3.4, we can see how the model looks for \( n \) amount of qubits. This model allows us to look at the differences in entropy entanglement when we choose the central qubit to be our system versus when we choose one of the environmental qubits to be our system. The Spin Bath Hamiltonian can be constructed according to:

\[
\hat{H}_{\text{SpinBath}} = \frac{1}{2} \sigma^z \otimes \sum_{i=1}^{n} g \cdot \sigma^z_i. \tag{3.7}
\]

Inherent in the 1D Ising Model’s structure lay the notion of the locality where only the nearest neighbors are coupled. This model was explored by Jordan Cotler et al. and was declared to have two tensor product structures in which locality emerges [3]. In Figure 3.4, we can see that the model resembles adding additional sides to a shape. We move from a line to a triangular structure, to a square, and so on. The 1D Ising Hamiltonian can be constructed according to:

\[
\hat{H}_{\text{Ising}} = J \cdot \sum_{i=1}^{n-1} \sigma^z_i \sigma^z_{i+1}. \tag{3.8}
\]

Once the Hamiltonians are constructed, we implement are newly added criterion: democracy of qubits. The goal of this criterion is to ensure that the selected TPS provides classicality for each qubit in the system. Thus, each qubit is put on equal
footing by allowing them to take turns being the "system". This is novel to this thesis as this criterion is not in any other Decoherence research. Rather than attacking this criterion from an initial state standpoint, we instead leave the initial states untouched, and we apply permutation matrices to $\hat{H}_{\text{native}}$.

$\hat{H}_{\text{native}}$ is constructed such that qubit 1 starts out as the "system". Once a permutation matrix is applied, qubit 2 becomes the "system", and qubit 1, along with other qubits, become the "environment". As seen in Figure 3.5, for the Spin Bath Model, this relegation of being the "system" has an effect on how the initial states are time-evolved because the dynamics of how the system is related to the environment changes. For the 1D Ising Model, all qubits are already on equal footing since no matter which qubit you label to be the system, each qubit is simply coupled to its nearest neighbors, and nothing changes. As a result of applying permutation matrices, once the Hamiltonians are constructed, a list of $n$ permutated matrices is sent to the next portion of the program to be scrambled.

Figure 3.4: Visualizations of Qubit Models: To the left the Spin Bath Model demonstrates coupling to a central qubit. To the right, the 1D Ising Model demonstrates locality with coupling to neighboring qubits.
Figure 3.5: Democracy of Qubits: The role of "system" gets relegated to each qubit so that the selected TPS demonstrates classicality for all qubits in the system. Qubit 1 starts out as the "system". This role gets passed to qubit 2 and so on.

### 3.3 Scrambling the Hamiltonian

The first step in scrambling the Hamiltonian is to construct a special unitary matrix $U$ that is characterized by $D^2 - 1$ real parameters and has $D^2 - 1$ traceless, Hermitian generators. Recall that $D = d_{sys} \cdot d_{env}$. The real parameters are defined as $\theta_{key}, \{\theta_{a}|a = 1, 2... (D^2 - 1)\}$, and the Hermitian generators can be identified with the Generalized Gell Mann Matrices (GGMMs), $\{\Lambda_{a}|a = 1, 2... (D^2 - 1)\}$. These matrices take the following three forms: symmetric, anti-symmetric, and diagonal matrices. The method for constructing GGMMs is further detailed in Appendix A and is taken from [1].

It can be shown that these tensor products from a basis for all hermitian traceless $D \times D$ matrices. If a system is divisible into sub-parts, one can find the GGMMs for each sub-part, and then tensor all of those GGMMs with each other. This method allows us to keep track of which GGMMs act on which sub-parts, as well as which
GGMMs correspond to interactions between sub-parts. Because we are working with qubits, each qubit is considered to be a sub-part of the composite system. The three GGMMs corresponding to one qubit come out to be the Pauli Sigma Matrices:

\[
\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.
\tag{3.9}
\]

Because all qubits have the same correspondent GGMMs, we are able to work directly with the Pauli Sigma Matrices to construct the composite system’s GGMMs. We can then create a master list that contains the order of how the matrices will be tensored with one another. To simplify this process, we can think of each qubit as having a certain slot in this list of orders. For instance, if we are working with a composite system of 2 qubits, there are 2 slots. The first slot is designated for qubit 1, and the second slot is designated for qubit 2. We can then associate each Pauli Sigma Matrix with a number. 1 for \(\sigma_x\), 2 for \(\sigma_y\), and 3 for \(\sigma_z\). Putting a number in one of the qubit slots signifies that the to-be-constructed GGMM will act on that qubit. Thus, we put a 0 in a qubit slot if the to-be-constructed GGMM will not act on that qubit. The 0 corresponds to the Identity matrix.

Putting all of this together, for 2 qubits we have the following list of matrices for the construction of the composite system’s GGMMs. Each list describes the order of tensoring.

<table>
<thead>
<tr>
<th>Q1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.1: GGMM Construction: Each column describes the tensoring order needed to construct a particular GGMM to create the complete set of GGMMs for a system of two qubits. Column 1 (1 0) corresponds to the tensoring order: \(\Lambda_1 = \sigma_x \otimes \hat{I}\).

Because the first three GGMMs have a 0 in Slot 2, we know that those first three GGMMs are qubit 1’s self-GGMMs extended to the proper dimension. The following three GGMMs correspond to qubit 2’s self-GGMMs, and the rest of the GGMMs correspond to interaction GGMMs. If we create a unitary matrix using just qubit 1’s self-GGMMs, only qubit 1’s basis is altered, and the Hamiltonian is scrambled to a new basis within the native TPS. In other words, Self-GGMMs allow us to move up and down a local unitary orbit. The GGMMs that alter the division between sub-parts and thus scramble a Hamiltonian to a new TPS are the interaction GGMMs.
Using this methodology to construct a list of GGMMs, we are able to construct
GGMMs for any \( n \) amount of qubits. Once we have the GGMMs, we can then
parameterize them with \( D^2 - 1 \) random float numbers defined as \( \theta_{\text{key}} \). The param-
eterized GGMMs are then summed, multiplied by \( i \), and exponentiated to form the
special unitary matrix\(^2\):

\[
\hat{U}_{\text{scram}} (\theta) = \exp \left( \sum_{n=1}^{D^2-1} i\theta_n \Lambda_n \right).
\] (3.10)

The final step of scrambling \( \hat{H}_{\text{native}} \) is to perform a unitary transformation on \( \hat{H}_{\text{native}} \)
using \( \hat{U} \):

\[
\hat{H}_{\text{scram}} = \hat{U}_{\text{scram}} (\theta)^\dagger \cdot \hat{H}_{\text{native}} \cdot \hat{U}_{\text{scram}} (\theta).
\] (3.11)

Each permutated matrix is scrambled using the same \( \theta_{\text{key}} \). The list of scrambled
Hamiltonians is then sent over to the gradient descent algorithm.

### 3.4 Gradient Descent

The list of scrambled Hamiltonians are sent over to the gradient descent algorithm to
optimize the thetas of the scrambling matrix \( \hat{U}_{\text{scram}} \) according to the cost function.
Gradient descent works by taking in some starting point described in terms of a
function’s parameters, and it minimizes the starting point using the gradient of a
function called the cost function. For each iteration, the algorithm updates the
parameters in the opposite direction of the gradient. The step size that determines
how much the parameters are updated is determined by the learning rate.

To give a more concrete example of how gradient descent works, let’s say our goal
is to minimize the function \( f ([x, y]) = \frac{1}{2}x^3 + \frac{1}{2}y^3 \). We give gradient descent the
random starting point of \([x, y] = [6, 2]\), which corresponds to a function value
of 112. Gradient descent will use the negative of the gradient of \(-\nabla f ([x, y]) =

\(^2\)Unlike for a general Lie group, the exponential map from algebra to group is surjective for
\( SU(D) \). Hence, we can be assured that as \( \theta \) ranges over all values, every element of \( SU(D) \) will be
parameterized.
− [3/2x^2, 3/2y^2], to determine the direction of fastest decrease of the cost function. It plugs the starting values into the gradient and multiplies this by the learning rate. If we were working with a learning rate of 1.0, we would have:

\[
1.0 \cdot \left( -\frac{3}{2} (6)^2, \frac{3}{2} (2)^2 \right) = [-54, -6].
\]

(3.12)

This vector is then added to the starting vector, and this process is continued until the minimum is found. Adapted from Mirko Stojiljkovic’s work on Real Python [8], the gradient descent algorithm we use looks like this:

```python
def gradient_descent(gradient, start, learn_rate, n_iter, cost_func, adjust, adjLR):
    '''Parameters:
    gradient: Python function that takes a vector and returns
    the gradient of the function we want to minimize.
    start: Starting vector.
    learn_rate: Learning rate that controls the magnitude of
    vector update
    n_iter: Number of iterations.
    cost_func: Function to be minimized.
    adjust: Iteration at which safety measure is implemented.
    adjLR: New learning rate implemented at adjust.
    
    Returns: Minimizing list of vectors.''
    
    theta_trial = start
    cost_history = np.zeros(n_iter)

    for i in range(n_iter):
        # Creating the vector that is equivalent to the "step"
        diff = -learn_rate * gradient(theta_trial)
        # Adding the step to the trial parameters
        theta_trial += diff
        # Computing the score of the new theta_trial
        score_tg = cost_func(theta_trial)
        # Keeping track of how the score changes
        cost_history[i] = score_tg

        # A safety measure that lowers the learning rate
        if i == adjust:
            learn_rate = adjLR

    print('Theta final guess score: ', cost_history[-1])
```

To ensure that the gradient descent algorithm settles at a true minimum, I have implemented two safety features. The first is the adjustment parameter that lowers the learning rate when the algorithm approaches the minimum. The second is a check of how the score transforms throughout the algorithm. As the algorithm runs, the algorithm keeps track of how the score changes with each iteration by saving the scores in a list defined as the cost history. Once the algorithm finishes running, the program takes a fit of the last 10 points in the cost history list. If the slope is less than \(-1 \times 10^{-5}\), the program continues the gradient descent algorithm where it left off. Thus, we have assured a minimum for every trial we run.

For the purposes of our research, our cost function must be able to take in a Hamiltonian and compute its entanglement entropy after a time evolution. Because gradient descent needs tangible variables to minimize, we use the list of thetas as the minimizing variables. The structure of the cost function will be detailed in the following section.

### 3.5 The Cost Function

The score we assign to a given TPS measures how much entropy is generated with the environment when the system starts out in an eigenstate of its best possible Candidate Pointer Observable. A high score would mean that no observable of the system can function as a Pointer Observable, corresponding to Case 3 in the Introduction. The cost function that we use has been modified over the course of our research. The first cost function allows us to minimize up and down a TPS line using gradient descent over the subset of thetas. The second cost function outputs a score for an entire TPS line as a minimum eigenvalue of a certain \(3 \times 3\) matrix, and thus reduces computational expense. I will detail both below.
**Cost Function 1**

The first cost function (CF1) begins by taking in a list of arbitrary theta values. These values are used to parameterize a special unitary matrix that scrambles the native Hamiltonians in the fashion described in Subsection 3.3. The initial state is then evolved forward in time according to each scrambled matrix. Recall that the initial states of the composite system are:

\[ \rho_{0,1} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \otimes \frac{1}{2^n} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{3.13} \]

\[ \rho_{0,2} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \otimes \frac{1}{2^n} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{3.14} \]

where Qubit 1 starts out in its pointer states, and Qubit 2 starts out in a maximally mixed ready state. The specific time the initial states are set to evolve to is the characteristic time, defined as:

\[ t_{\text{char}} = \frac{1}{\|\hat{H}\|_2}, \tag{3.15} \]

with \(\|\hat{H}\|_2\) a matrix norm, as described in the Background Section. Since \(\|\hat{U} \cdot \hat{H} \cdot \hat{U}^\dagger\| = \|\hat{H}\|\), \(t_{\text{char}}\) gives a good common benchmark time. With the time-evolved state, we can now follow the methodology of the qubit example in Subsection 2.2 and reduce the density matrix to compute the purity entropy. The purity entropy tells us how much entanglement entropy was generated over the time evolution, serving as a measure of how resistant the system was to decoherence. Because each permutated Hamiltonian evolves 2 initial states forward in time, we end up with a total of \(2n\) scores. These scores are averaged to give us the final score.

Summarizing CF1 with a series of qualitative steps:

<table>
<thead>
<tr>
<th>Cost Function 1 ((\theta_{\text{Arbitrary}}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Parameterize GGMMs to create a specialized unitary matrix (U)</td>
</tr>
<tr>
<td>2. Perform unitary transformation on (H_{\text{scram}})</td>
</tr>
<tr>
<td>3. Evolve all initial states forward in time according to (H_{\text{scram}})</td>
</tr>
<tr>
<td>4. Reduce the evolved state</td>
</tr>
<tr>
<td>5. Compute purity entropy of reduced state</td>
</tr>
<tr>
<td>6. Average all scores to get the final score</td>
</tr>
</tbody>
</table>
CF1 takes in a set of arbitrary thetas and outputs a score. However, it only computes the score of a particular basis within a Tensor Product Structure, not the score that represents the entire Tensor Product Structure. Recall that for the space of Tensor Product Structures, we can imagine each TPS as a line. Although all the possible bases in that line share the same division between system and environment, each basis still varies in score. For each line, there is a minimum score that represents the total score of the entire TPS. Thus, for CF1 to compute the score of a TPS, it has to use gradient descent to minimize individual basis scores. CF1 was used in the first half of the research, but because it is not able to immediately compute the score of a TPS, the gradient descent algorithm moves up and down TPS lines, as well as across TPS lines. This results in increased computational expense, and thus we were motivated to find an alternative method of computing Tensor Product Structure scores.

**Cost Function 2**

The following description of the second iteration of our cost function is closely excerpted from the notes of [7] (recorded in preparation for a future publication). The method CF2 was derived by Dr. Setter and implemented in code by myself.

The improved cost function (which we dub CF2) operates similarly to CF1 above and, in fact, we perform explicit checks in Section 4 to show that they output the same numbers. However, CF2 scores a TPS as a whole not by performing a search, but, rather, by recognizing that the function to be minimized turns out to be a certain quadratic form and therefore its minimization can be performed by computing a matrix eigenvalue. (This minimization is over just one "line" in Figure 5.2 and gradient descent is still necessary to move from one line to the next.) The analytic formulas we derive below exploit the fact that our system is a qubit and represent one of the central results of our research.

For the purposes of writing our analytic formula in the most general way, we again take our "system" to be qubit 1, with dimension $d_1 = 2$, while the $n$ remaining qubits are grouped together as "environment", thought of as a single sub-factor of dimension $d_2 = 2^n$. The total tensor product space $\mathcal{H}$, therefore, has dimension $D = 2 \cdot d_2$. As described above for CF1, the aim of the cost function is to assign a score to a traceless, hermitian, $2d_2 \times 2d_2$ matrix $\hat{H}$, which we, therefore, treat as a given. (Also, as emphasized in Section 2.3 the particular matrix representation $\hat{H}$ implicitly defines a certain splitting of $\mathcal{H}$ into system and environment, which
splitting we are testing out for the existence of a good pointer observable.)

Recall that a good pointer observable will exist if there is some complete, orthogonal basis of initial states for the system that do not generate much entropy after time evolution (i.e. good eigenstates for the pointer observable). The advantage of our system being a qubit is that, for a given initial state (denoted $|+n\rangle$), there is only one other orthogonal state (denoted $|-n\rangle$) up to an irrelevant phase: states and orthogonal bases are in 1-1 correspondence. Further, the set of possible initial states can be easily parameterized and visualized using the Bloch Sphere representation of the states of a qubit, see Figure 3.6. In this visualization, an orthogonal basis $\{|+n\rangle, |-n\rangle\}$ corresponds to a pair of antipodal points on the Bloch Sphere of the system qubit. The idea is to write an analytic formula for the entropy generated when the system starts out in the two states $|\pm n\rangle$ and the environment starts in a maximally mixed ready state. Then, to use this formula to compute the minimum over all initial states. This minimum becomes the score for $S(\hat{H})$ for $\hat{H}$: how good its best pointer observable is.

![Bloch Sphere Diagram](image)

Figure 3.6: Initial States Visualized. Environment starts in maximally-mixed ready state and system starts out in orthogonal pair of states $|\pm n\rangle$.

In fact, this gives a score for the entire local unitary (LU) orbit of $\hat{H}$: an LU of the form $\hat{U}_1 \times \hat{U}_2$ acting on $\hat{H}$ has the effect simply of rotating the system Bloch sphere (since the environmental ready state has the convenient property of being invariant under $\hat{U}_2$). Hence, the minimum $S(\hat{H})$ (before rotation) is the same as $S(\hat{H}')$ (after rotation) for any LU-related $\hat{H}'$.

We describe the foregoing in more detail and then derive analytic formulae.
Up to irrelevant phases, each pair of orthogonal states $|\pm n\rangle$ of a qubit are labeled by a unit vector $\vec{n}$ in 3d space or a set of two coordinates $(\theta, \phi)$ ("latitude, longitude"):

\[ \vec{n} = (\cos(\phi) \sin(\theta), \sin(\phi) \sin(\theta), \cos(\theta)) \]  
\[ |+n\rangle = \cos\left(\frac{\theta}{2}\right) |+z\rangle + e^{i\phi} \sin\left(\frac{\theta}{2}\right) |-z\rangle \]  
\[ |-n\rangle = \sin\left(\frac{\theta}{2}\right) |+z\rangle - e^{i\phi} \cos\left(\frac{\theta}{2}\right) |-z\rangle \]

where we denote $|+z\rangle = |0\rangle$, $|-z\rangle = |1\rangle$ since we are visualizing the abstract qubit states as being situated in space in our Bloch Sphere representation. The corresponding pure density matrices are given by the convenient formula:

\[ \rho_{CPO}^{\pm} = |\pm n\rangle \langle \pm n| = \frac{1}{2} \left( \hat{I} \pm \vec{n} \cdot \vec{\sigma} \right) \]  

where CPO stands for candidate pointer observable and $\vec{\sigma} = (\sigma_1, \sigma_2, \sigma_3)$ are the three Pauli sigma matrices and $\hat{I}$ is the $2 \times 2$ identity. In line with discussion of CF1, we have the following initial states of the composite system:

\[ \rho^{(\pm)}_0 = \left( \rho_{CPO}^{(\pm)} \right) \otimes \left( \frac{1}{d_2} \hat{I} \right) \]  

where now $\hat{I}$ is the $d_2 \times d_2$ identity. Here is the sequence of steps:

1. Evolve each initial state with Hamiltonian $\hat{H}$ by characteristic time, $\tau = t_{\text{char}} = 1/||\hat{H}||$. Explicitly,

\[ \rho^{(\pm)}_\tau = e^{i\tau \hat{H}} \rho^{(\pm)} \rho^{(\pm)} e^{-i\tau \hat{H}} \]  
\[ = \frac{1}{2d_2} \hat{I} \otimes \hat{I} \pm \frac{1}{2d_2} \sum_{a=1}^{3} n_a e^{i\tau \hat{H}} (\sigma_a \otimes \hat{I}) e^{-i\tau \hat{H}} \]  

2. Partial trace over all but qubit 1: $\left( \rho^{(\pm)}_\tau \right)_1 = \text{Tr}_2 \left( \rho^{(\pm)}_\tau \right)$. Explicitly, define the following three matrices:

\[ Q_\alpha = \frac{1}{d_2} e^{i\tau \hat{H}} (\sigma_\alpha \otimes \hat{I}) e^{-i\tau \hat{H}}, \quad \text{for } \alpha = 1, 2, 3 \]  

We then have:

\[ \left( \rho^{(\pm)}_\tau \right)_1 = \text{Tr}_2 \left( \rho^{(\pm)}_\tau \right) = \frac{1}{2} \hat{I} \pm \frac{1}{2} \sum_{a=1}^{3} n_a (\text{Tr}_2 Q_\alpha) \]
3. Compute purity entropies of these reduced density matrices:

\[ S^{(\pm)}(\vec{n}) = 1 - \text{Tr} \left[ \left( \rho^{(\pm)}_\tau \right)_1 \cdot \left( \rho^{(\pm)}_\tau \right)_1 \right] \]

Squaring the expression (3.24):

\[ \left[ \left( \rho^{(\pm)}_\tau \right)_1 \right]^2 = \frac{1}{4} \pm \frac{1}{2} \sum_{a=1}^{3} n_a (\text{Tr}_2 Q_\alpha) + \frac{1}{4} \sum_{\alpha, \beta=1}^{3} n_\alpha n_\beta \text{Tr}_2 (Q_\alpha) \cdot (Q_\beta) \]

(3.25)

\[ \Rightarrow \text{Tr} \left[ \left( \rho^{(\pm)}_\tau \right)_1 \right]^2 = \frac{1}{2} \pm \frac{1}{2} \sum_{a=1}^{3} n_a \text{Tr} [(\text{Tr}_2 Q_\alpha)] + \frac{1}{4} \sum_{\alpha, \beta=1}^{3} n_\alpha n_\beta \text{Tr} \left[ (Q_\alpha) \cdot (Q_\beta) \right] \]

(3.26)

However, using one of a useful list of partial trace identities compiled in reference [10], we have \( \text{Tr} [(\text{Tr}_2 Q_\alpha)] = \text{Tr} Q_\alpha = \text{Tr} (\sigma_\alpha \otimes I) = 2 \text{Tr} \sigma_\alpha = 0 \) since the Pauli sigma matrices are traceless. That is, the middle term in (3.26) vanishes identically. The purity entropy becomes:

\[ S^{(\pm)}(\vec{n}) = \frac{1}{2} - \frac{1}{4} \sum_{\alpha, \beta=1}^{3} n_\alpha n_\beta \text{Tr} \left[ (Q_\alpha) \cdot (Q_\beta) \right] \]

(3.27)

Interestingly, the initial states \(|+n\rangle\) and \(|-n\rangle\) turn out to generate the same purity entropy after time evolution. This fact, which we confirmed numerically, is special to dimension two: if the dimension of the system were larger than two, then different elements of an orthogonal basis of system states would in general generate different amounts of entropy.

4. Average: \( S(\vec{n}) = \frac{1}{2} \left[ S^{(+)}(\vec{n}) + S^{(-)}(\vec{n}) \right] = S^{(+)}(\vec{n}) \). This is trivial since we have just noted that \( S^{(+)}(\vec{n}) = S^{(-)}(\vec{n}) \). We include this step to make contact with CF1 and in anticipation of choosing larger system sizes for future research.

5. Minimize: \( S(H) = \min_{\vec{n}} S(\vec{n}) \), where the minimization is over all unit vectors \( \vec{n} \) with \( |\vec{n}| = 1 \). Note that the second term in the expression (3.27) is a quadratic form in the three components of \( \vec{n} \). Expressible as the matrix multiplication

\[ S^{\pm}(\vec{n}) = \frac{1}{2} - \frac{1}{4} \vec{n}^T \cdot M \cdot \vec{n} \]

(3.28)

where \( M \) is the real, symmetric, \( 3 \times 3 \) matrix with entries:

\[ M_{\alpha \beta} = \text{Tr} \left[ (\text{Tr}_2 Q_\alpha) \cdot (\text{Tr}_2 Q_\beta) \right] \]

(3.29)
Therefore, using a standard result of linear algebra, the optimal candidate pointer observable is when \( \vec{n} \) is a normalized eigenvector of \( M \) with the largest eigenvalue \( \lambda \). The corresponding score is \( S(\hat{H}) = \frac{1}{2} - \frac{1}{4}\lambda \).

This concludes the close excerpting of [7]. Taking stock, we summarize the cost function CF2 as a series of computational steps replacing a gradient descent-type search over a local unitary orbit:

**Cost Function 2 (\( \theta_{\text{Arbitrary}} \))**:
1. Using Pauli Sigma Matrices and environment ready state, compute \( Q_\alpha = (Q_1, Q_2, Q_3) \)
2. For each pair of \( Q_\alpha \) (i.e. 11, 12, 13, 22...)
   a. Reduce both pairs
   b. Compute the dot product of the reduced pairs
   c. Take the trace of the result
3. Place the computed values in the corresponding Matrix slots (i.e. Pair 11 = Element \( M_{\alpha,\beta} \{1\}[1] \))
4. Compute the largest eigenvalue \( \lambda \) of \( M_{\alpha,\beta} \)
5. The score will be \( S = \frac{1}{2} - \frac{1}{4}\lambda \)
6. Average all scores to get the final score

Unlike CF1, CF2 computes the score of the entire tensor product structure. This greatly reduces computational expense because now the gradient descent algorithm mainly moves from TPS line to TPS line as opposed to also looking up and down each TPS line.

### 3.6 Local Unitary Orbit Checks

Using the cost functions above, gradient descent finds a TPS in which the minimum entropy entanglement of 0 is generated. Because the goal of this research is to see if the newfound basis is in the native TPS or an entirely new TPS, we must verify whether or not the new found basis shares a local unitary (LU) orbit with \( \hat{H}_{\text{native}} \).
To do this, we turned to work done by Sun et al. in which Sun and collaborators investigated the invariants under local unitary transformations for two and three qubit systems [9]. Indeed the availability of these math results is one of the reasons for the focus on qubits and is a major contribution to the present work. Their research is geared around density matrices, however, we generalized their work to be inclusive of Hamiltonians since both operators are $2^n \times 2^n$ traceless, Hermitian matrices and examination of their proofs show that the trace of the matrices play no role (density matrices have trace 1, while $\hat{H}$ has trace 0. The first key to this generalization is that the Hamiltonian can be expressed as:

$$\hat{H} = \sum_{\alpha=1}^{3} h_{\alpha,0} (\sigma_{\alpha} \otimes \hat{I}) + \sum_{\alpha=1}^{3} h_{0,\alpha} (\hat{I} \otimes \sigma_{\alpha}) + \sum_{\alpha,\beta=1}^{3} h_{\alpha,\beta} (\sigma_{\alpha} \otimes \sigma_{\beta}). \quad (3.30)$$

The coefficients $h_{\alpha,0}$, $h_{0,\alpha}$, and $h_{\alpha,\beta}$ are used in the value checks, thus we must derive what those values are. We can do this by computing:

$$h_{\alpha,0} = Tr \left[ (\sigma_{\alpha} \otimes \hat{I}) \cdot \hat{H} \right]. \quad (3.31)$$

Recall that the self-GGMMs of qubits are simply the Pauli sigma matrices. Thus, we are really computing:

$$h_{\alpha,0} = Tr [\Lambda_{\alpha} \cdot \hat{H}]. \quad (3.32)$$

These values are used to form the following vectors and matrix that are used in the value checks:

$$h_{\alpha,0} \rightarrow h_1 = [h_{10}, h_{20}, h_{30}], \quad (3.33)$$

$$h_{0,\alpha} \rightarrow h_2 = [h_1, h_2, h_3], \quad (3.34)$$

$$h_{\alpha,\beta} \rightarrow h_{\text{int}} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}. \quad (3.35)$$

To verify whether or not two Hamiltonians $\hat{H}_1$ and $\hat{H}_2$ are a part of the same LU orbit for systems of 2 qubits, i.e.:

$$H_2 = (U_1 \otimes U_2) \cdot H_1 \cdot \left( U_1^\dagger \otimes U_2^\dagger \right). \quad (3.36)$$

We can compute two series that include values needed to compute the final checks. The series are as follows:

$$S_1 = \{ \tilde{h}_1, [h_{\text{int}}] \tilde{h}_2, [h_{\text{int}}][h_{\text{int}}]^T \tilde{h}_1, [h_{\text{int}}][h_{\text{int}}]^T \tilde{h}_2, \} \quad (3.37)$$

$$\tilde{h}_j^T \left( [h_{\text{int}}][h_{\text{int}}]^T \right)^2 \tilde{h}_1, \tilde{h}_j^T \left( [h_{\text{int}}][h_{\text{int}}]^T \right)^2 [h_{\text{int}}] \tilde{h}_2, \} \quad (3.38)$$
\[ S_2 = \{ \tilde{h}_2, [h_{int}]^T \tilde{h}_1, [h_{int}]^T [h_{int}] \tilde{h}_2 \}. \]  
(3.39)

For sake of simplicity, we can express each series in terms of the variables \( \mu \) and \( \nu \):

\[ S_1 = \{ \tilde{\mu}_1, \tilde{\mu}_2, \tilde{\mu}_3, \ldots \}, \]  
(3.40)

\[ S_2 = \{ \tilde{\nu}_1, \tilde{\nu}_2, \tilde{\nu}_3, \ldots \}, \]  
(3.41)

where \( \tilde{\mu}_n \) and \( \tilde{\nu}_n \) just correspond to the elements listed in Equations 3.34 and 3.35. We are now finally able to compute the value checks! The value checks are as follows:

\[ \langle \tilde{\mu}_n, \tilde{\mu}_n \rangle, \ n = 1, 2, 3, \]  
(3.42)

\[ \langle \tilde{\nu}_n, \tilde{\nu}_n \rangle, \ n = 1, 2, 3, \]  
(3.43)

\[ \langle \tilde{\mu}_1, \tilde{\mu}_2, \tilde{\mu}_3 \rangle, \ n = 1, 2, 3, \]  
(3.44)

where \( \langle ..., ... \rangle \) = dot product. For \( \tilde{H}_1 \) and \( \tilde{H}_2 \) to share the same LU orbit, they must be invariant under every single value check. These checks are only for systems of 2 qubits, however, a similar process is used to derive the checks for systems of 3 qubits.

### 3.7 The Complete Python Program

Now that each sub-part of the program has been fleshed out, we can detail the full process of the most updated version of the python program.

1. **The Program Takes In:**
   - (1) \( nqs \) = a specified number of environmental qubits
   - (2) \( hamnum \) = a Hamiltonian number
   - (3) \( cc \) = a coupling coefficient
   - (4) \( its \) = iterations for gradient descent
   - (5) \( lrate \) = learning rate for gradient descent
   - (6) \( adj \) = adjusted iterations
   - (7) \( newlrate \) = adjusted learning rate
   - (8) \( runnum \) = the number of times to run
2. **Parameter 1, *nqs*, is used to:**
   a. Document the dimensions of the composite system:
   \[
   d_{\text{sys}} = 2, \quad (3.45) \\
   d_{\text{env}} = 2^n, \quad (3.46) \\
   D_H = d_{\text{sys}}d_{\text{env}}. \quad (3.47)
   \]

   b. Construct one of the four native Hamiltonians, specified by Parameter 2, *hamnum*:
   \[
   \text{hamnum}(1) = \text{Spin Bath} \\
   \text{hamnum}(2) = \text{1D Ising} \\
   \text{hamnum}(3) = \text{C-NOT} \\
   \text{hamnum}(4) = \text{3D Heisenberg}
   \]
   \[
   \hat{H}_{\text{native}} = \text{hamnum}(n)
   \]

   c. Construct the environment’s initial state:
   \[
   \rho_{0,\text{env}} = \frac{1}{d_2} \hat{I}. \quad (3.48)
   \]

3. **Parameter 3, *cc*, is used to:**
   Determine how coupled each qubit is to its neighbors. Although we currently have the coupling coefficient as an adjustable parameter, we are consistently using a coupling coefficient of 1. Varying the coefficient would add a whole other layer of depth to this research, and our goal is to just lay the groundwork.

4. **Democracy of Qubits:**
   Implementing our new criterion using *nqs* permutation matrices, we permute \( \hat{H}_{\text{native}} \) to give us a list of Hamiltonians that put each environmental qubit on equal footing with qubit 1. This is also part of the definition of a TPS.

5. **Scrambling the Hamiltonian:**
   We construct \( \theta_{\text{key}} \) using \( D^2 - 1 \) random float numbers. We use \( \theta_{\text{key}} \) and the list of GGMMs to scramble \( \hat{H}_{\text{native}} \) to \( \hat{H}_{\text{scrambled}} \):
   \[
   \theta_{\text{key}} = \left\{ \theta_a | a = 1, 2, \ldots, D^2 - 1 \right\}, \quad (3.49) \\
   \hat{U}_{\text{scram}} (\theta_{\text{key}}) = \exp \left( \sum_{n=1}^{D^2-1} i\theta_{\text{key},n} \Lambda_n \right), \quad (3.50)
   \]
\[ \hat{H}_{\text{scram}} = \hat{U}_{\text{scram}}(\theta_{\text{key}})^\dagger \cdot \hat{H}_{\text{unscram}} \cdot \hat{U}_{\text{scram}}(\theta_{\text{key}}) \cdot \] (3.51)

6. **Preliminary Gradient Descent:**
Algorithm defines \( \theta_0 = D^2 - 1 \) random float numbers, CF2, the gradient of CF2, and then turns to the gradient descent algorithm.

7. **Gradient Descent:**
The gradient descent algorithm puts parameters 4,5,6,7 to use where 4 and 5 are used to determine the initial iterations and learning rate, and 6 and 7 determine the point at which the learning rate is lowered to ensure that the algorithm settles at a minimum. It takes in \( \theta_0 \) to function as the starting point of the algorithm, and then it works to unscramble the Hamiltonians in \( \hat{H}_{\text{scram}} \) one by one. \( \theta_0 \) is used to parameterize a unitary matrix that is applied to \( \hat{H}_{\text{scram}} \). This gives us the general “\( \hat{H} \)” that is then used for the rest of CF2’s scoring process. Gradient descent plugs \( \theta_0 \) into CF2’s gradient, computes the direction in which the theta values produce a lower score, and then adds these values to \( \theta_0 \). This process repeats until the score settles at or really close to 0.

8. **Local Unitary Checks:**
Using the winning set of theta’s, which we define as \( \theta_{qc} \), we construct the Hamiltonian that is representative of the basis that gradient descent found. Recall that gradient descent was working to find a set of theta values that minimized the \( \hat{H}_{\text{scram}} \). Thus, for each Hamiltonian in \( \hat{H}_{\text{scram}} \), \( \theta_{qc} \) is used to construct a specialized unitary matrix that is then applied to \( \hat{H}_{\text{scram}} \). The set of “unscrambled” Hamiltonians are defined as \( \hat{H}'_{\text{scram}} \). These are the Hamiltonians that are sent over to the LU checks to be checked alongside \( \hat{H}_{\text{native}} \). When we get to the LU checks, the module takes in \( \hat{H}_{\text{native}} \) and \( \hat{H}'_{\text{scram}} \). It computes all of the invariant values for both Hamiltonians separately and then lines up each value next to each other for comparison. From this point, we are able to see whether or not the basis we found exists in the same local unitary orbit/tensor product structure as \( \hat{H}_{\text{native}} \), or if we have discovered a new basis that exhibits classical behavior.
In this chapter, I will detail the methods used to check that the algorithm was working as intended.

4.1 Testing Gradient Descent

To verify that the gradient descent algorithm was working properly, it was tested on the smaller subset of \(\theta\)'s that are used to parameterize the specialized unitary matrix involved in scrambling. Recall that the smaller subset limits gradient descent to minimizing up and down a tensor product structure line. For a given \(\hat{H}\), CF2 lets us compute the score of the optimal candidate pointer observable in \([\hat{H}]\) as eigenvalues of a matrix. Because CF2 eradicates the option of minimizing up and down a TPS line, we tested gradient descent using CF1.

We tested the native Tensor Product Structure since in this particular case, we can predict the results of gradient descent. Within the native TPS, we should expect a minimum score of 0 since this is where there exists a well-defined pointer observable. Using the cost history feature that I implemented within the gradient descent algorithm, we were able to see how the score of the system changed throughout the iterations of gradient descent. For nearly all of the trials, we observed a graph with the trend seen in Figure 4.1.

We can see that the cost history settles nicely at 0, as expected. This allowed us to confirm that the gradient descent algorithm was effective in finding minimum scores.

4.2 Testing Local Unitary Checks

To confirm that the Local Unitary checks were working properly, we decided to check two bases within a random tensor product structure. Rather than using the Hamiltonians that represent the qubit systems we are exploring (as they hold some structure), we instead constructed a completely randomized Hermitian matrix. We then scrambled the randomized Hermitian matrix using qubit1’s self-GGMMs, giving us two matrices to test. Because we only used the subset of \(\theta\)'s to scramble, we know that both Hamiltonians lay in the same TPS. For the checks to work, every
Figure 4.1: Cost History for $\hat{H}_{\text{native}}$: When minimizing up and down a TPS line using the native Hamiltonian as a starting point, the score settles at 0. This shows that in the native TPS, classical behavior emerges as we are able to achieve 0 entanglement entropy growth.

A single value check must be invariant for both Hamiltonians.

<table>
<thead>
<tr>
<th>Value</th>
<th>for H1:</th>
<th>for H2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(-0.01684901-0.01928891j)</td>
<td>(-0.01684901-0.01928891j)</td>
</tr>
<tr>
<td>1</td>
<td>(-0.00939418-0.01521632j)</td>
<td>(-0.00939418-0.01521632j)</td>
</tr>
<tr>
<td>2</td>
<td>(0.0043991-0.00061899j)</td>
<td>(0.0043991-0.00061899j)</td>
</tr>
<tr>
<td>3</td>
<td>(-0.00032374+0.00070467j)</td>
<td>(-0.00032374+0.00070467j)</td>
</tr>
<tr>
<td>4</td>
<td>(-0.00029586+0.0010656j)</td>
<td>(-0.00029586+0.0010656j)</td>
</tr>
<tr>
<td>5</td>
<td>(-0.00012778-0.00014399j)</td>
<td>(-0.00012778-0.00014399j)</td>
</tr>
</tbody>
</table>
As seen above, each value of the checks was the same for both Hamiltonians, confirming that the Local Unitary checks work seamlessly for two qubits.

4.3 Testing CF2

To verify that the new cost function was outputting the same score found when CF1 was used to minimize up and down a local unitary orbit line, we compared scores for random Hamiltonians.

As seen in Figure 4.2, our check confirmed that CF2 produced accurate scores.
Our first result came from the exact method we used to check that gradient descent was working as intended. Instead of feeding $\hat{H}_{\text{native}}$ into the algorithm to minimize down a TPS line, we fed in a random $\hat{H}_{\text{scram}}$. We found that nearly all $\hat{H}_{\text{scram}}$s would settle at a non-zero value. This shows that some TPSes are better than others at giving an account of decoherence because even when minimizing down a TPS line, we weren’t able to reach 0 entanglement entropy growth. This confirms the proposal of Carroll and Singh in that a well-defined pointer observable might distinguish TPSes [1].

We are able to go beyond [1] in performing a search over the entire space of
TPSes. When we ran gradient descent using all theta parameters, we were able to successfully settle at a minimum. Thus, our quantitative criterion was successful in selecting a preferred tensor product structure. When put through the local unitary checks, we found that for the bases that we found, not all values of the checks were equivalent. This means that we found bases outside of the native local unitary orbit that demonstrated classical behavior. Thus, for two qubits, the preferred TPS appears to be not unique.

5.2 The Three Qubit Case

In line with the results for 2 qubits, the $\hat{H}_{scram}$s for 3 qubits settled at non-zero values when put through gradient descent using only the subset of theta parameters. We have yet to verify whether or not the newfound bases are a part of the same LU orbit as $\hat{H}_{native}$. We suspect that higher numbers of qubits may admit unique TPSes, but we leave this for future research.
Chapter 6

CONCLUSION AND FUTURE DIRECTIONS

In this thesis, I have coded a Python Program that takes in a random Tensor Product Structure, looks through the space of TPSes, and successfully finds a TPS in which classical behavior emerges. We confirmed Carroll, Singh [1], and Tegmark’s [10] proposal that a well-defined pointer observable might select a preferred TPS. We created a new criterion to be considered: democracy of qubits, which relegates the role of "system" to each environmental qubit, ensuring that the selected TPS demonstrates classical behavior for all qubits. We generalized Sun et al.’s [9] work to be inclusive of Hamiltonians and translated it into working code. This allowed us to confirm whether or not two Hamiltonians belonged to the same TPS. All of these results came from the code that I wrote, displayed in Appendix B.

Although for 2 qubits we were able to conclude that the preferred TPS was not unique, this result was expected due to the small dimensionality of the composite system. Preliminary results suggest that for the 3 qubit case, there may not be as many TPSes that demonstrate classical behavior due to the higher dimensionality of the composite system.

With the code generalized to work for \( n \) amount of qubits, one could begin to explore systems of higher dimension. In order to start exploring systems with \( n > 3 \) qubits while maintaining minimal computational expense, the program would need to be optimized. For systems of two and three qubits, the program runs at a reasonable speed, but once we hit higher numbers of qubits, it can take a considerable amount of time to settle at a minimum. One idea for optimization is to derive the gradient of the cost function by hand so that python doesn’t have to use numerical methods when running gradient descent. Another idea is to use a gradient descent algorithm that comes directly from a machine learning package. With the many weights of machine learning algorithms, this could greatly speed up the process of gradient descent and would eradicate the need for the numerous safety measures I’ve put in place. We could also turn to a gradient descent method of Stochastic Gradient Descent which would allow the algorithm to search over the space of TPSes more quicker. Furthermore, for systems of two and three qubits, we are able to use the local unitary checks formalized by Sun et al. However, there is no literature yet on
how to check local unitary invariance for systems with four or more qubits. These value checks would need to be derived to verify if the new-found bases reside in the native TPS.

We plan on cross-checking our results with the findings of Cotler et al. [3] to see how locality plays into the emergence of classicality. There is still more to analyze, and a variety of directions this research can go. One direction is looking at the TPSes that the algorithm found, and seeing if there are any patterns within the Hamiltonians (i.e. scalar multiples, period differences, etc.). There can be analysis done on how results for qubit 1 differ from the environmental qubits. We can explore systems with varying coupling strengths, as well as systems with active self-Hamiltonians. The code allows for all of this exploration, thus using this code, one can simply alter a few parameters and develop new methods of analysis. We have seen already with the simplest possible system of two qubits that some ways of dividing into "system" and "environment" allow for the existence of a well-defined pointer observable while others do not. For more complicated systems, it may turn out that there is a (nearly) unique way of doing so, going on a long way toward explaining why the world appears the way it does to us, classical observers.


CONSTRUCTION OF GENERALIZED GELL-MANN MATRICES

Following Carroll and Singh [1], to scramble a Hamiltonian, we first construct a special unitary matrix $U$ that is characterized by $D^2 - 1$ real parameters and has $D^2 - 1$ traceless, Hermitian generators, where $D = d_{sys} \cdot d_{env}$. The real parameters are defined as $\theta_{\text{key}, a = 1, 2... (D^2 - 1)}$, and the Hermitian generators can be identified with the Generalized Gell Mann Matrices (GGMMs), \{\Lambda_{a} | a = 1, 2... (D^2 - 1)\}. These matrices take the following three forms: symmetric, anti-symmetric, and diagonal matrices. The equations for constructing each form are as follows:

\[
\Lambda^{jk}_{\text{sym}} = E^{kj} + E^{jk}, \quad 1 \leq j < k \leq D \quad (A.1)
\]

\[
\Lambda^{jk}_{\text{antisym}} = -i \left( E^{jk} - E^{kj} \right), \quad 1 \leq j < k \leq D \quad (A.2)
\]

\[
\Lambda^{jk}_{\text{antisym}} = -i \left( E^{jk} - E^{kj} \right), \quad 1 \leq j < k \leq D \quad (A.3)
\]

Where $E^{jk}$ is a $D \times D$ matrix with all zeros except for a 1 in the $(j, k)$ location.

Because we are working with systems of qubits, we can compute the GGMMs for each qubit individually. Following the equations above, the GGMMs for a qubit where $d = 2$ turn out to be the three Pauli Sigma Matrices. Thus for a system of $n$ qubits, we have $\Lambda_{qu1}, \Lambda_{qu2},...\Lambda_{qu_n}$. The complete set of GGMMs for the composite system would therefore be every matrix in each set tensored with each other.
CODE IMPLEMENTATION

Here I will display all of the code used to produce the above results. The code is organized in a manner where a single file is used to run continuous trials of finding a preferred TPS. Thus, all of the code used to construct Hamiltonians, initial states, and other such things are organized in modules. Each module will be displayed in the order in which they are introduced in this thesis. The last section of the code will be the running file.

```python
# Constructing Hamiltonian 1
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# The main function of this module is to construct the
# Hamiltonian found in the Decoherence text
# -- this Hamiltonian describes a system where one qubit is
# coupled with n other qubits
# -- note: this system has no self-Hamiltonians
# -- takes in two variables:
# ----> n: number of environmental qubits
# ----> g: strength of coupling
# -- outputs H1
#

import numpy as np

# Creating the needed Matrices
sigma_z = np.array([[1, 0],
                    [0, -1]])
Id = np.identity(2)

def construct_H1(n, g):
    '''Parameters:
    n: Number of Qubits.
    g: Coupling coefficient.

    Returns: Spin Bath Hamiltonian.''
    H_tot = np.zeros((2**(n+1), 2**(n+1)))
```
# Setting the first slot of every term (aka the system)
term = 1/2*sigma_z

# i corresponds to term number
for i in range (1, n+1):
    # j corresponds to slot in term
    for j in range (1, n+1):
        # term x Identity unless i = j
        if j < i or j > i:
            term = np.kron(term, Id)
        # i = j, term x sigma_z
        else:
            term = np.kron(term, g*sigma_z)
        # add term to Hamiltonian
        H_tot += term
    # reset term
    term = 1/2*sigma_z

print("H1: ", H_tot)
return(H_tot)

# Constructing Hamiltonian 2
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
### The main function of this module is to construct the 1-D Ising Hamiltonian found in the Cotler text
### -- this Hamiltonian describes a system where one qubit is coupled with n other qubits in a chain-like, circular manner
### -- for example, if n=2, you can imagine a triangular chain,
### or if n=4, you have a square chain
### -- note: this system has self-Hamiltonians
### -- takes in three variables:
### ----> n: number of coupled qubits
### ----> J: strength of coupling
### ----> h: strength of self-Hamiltonian -- predefined as 0 for now
### -- outputs H2
###
import numpy as np

# Creating the needed Matrices
51

sigma_z = np.array([[1, 0],
                    [0, -1]])

sigma_x = np.array([[0, 1],
                    [1, 0]])

Id = np.identity(2)

# Although this Hamiltonian includes self-Hamiltonians, for
# simplicity, we will start with:
# h = 0

def construct_H2(n, J):
    '''Parameters:
    n: Number of environmental Qubits.
    J: Coupling Coefficient

    Returns: Ising Model Hamiltonian.''

    H_tot = np.zeros((2**(n+1), 2**(n+1)))

    ### ADDING IN SELF-HAMILTONIANS ###
    term = h*sigma_x
    # i corresponds to term number
    for i in range (0, n+1):
        # j corresponds to slot in term
        for j in range (0, n):
            if (j+1) == i:
                term = np.kron(term, sigma_x)
            # all other terms are tensored with identity
            else:
                term = np.kron(term, Id)
            # add term to Hamiltonian
        H_tot += term
    # after first term, first tensor changes to identity
    term = h*Id

    ### ADDING IN INTERACTION HAMILTONIANS ###
    # First making the list of matrices to be tensored
    kronlist = [sigma_z, sigma_z]
    for j in range (1, n):
        kronlist.append(Id)

    # [term, term] // n = 1 ---> run 1 times
# [term, term, id] // n = 2 ---> run 3 times
# [term, term, id, id] // n = 3 ---> run 4 times

if n>1:
    # Runs number of total qubits times...
    for i in range(0, n+1):
        term = kronlist[0]
        # Runs number of env qubits times
        for k in range(0, n):
            term = np.kron(term, kronlist[k+1])
        H_tot += -J*term
        # Changing order of list
        kronlist.insert(0, (kronlist.pop(n)))
else:
    term = kronlist[0]
    # Runs number of env qubits times
    for k in range(0, n):
        term = np.kron(term, kronlist[k+1])
    H_tot += -J*term

print("H2: ", H_tot)
return(H_tot)

#################################################################
### Constructing Hamiltonian 3
### ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
### The main function of this module is to construct the C-NOT
### -- takes in three variables:
### ----> n: number of environmental qubits
### ----> c: strength of coupling
### -- outputs H3
###

import numpy as np

# Creating the needed Matrices
state0 = np.array([[1, 0],
                   [0, 0]])
state1 = np.array([[0, 0],
                   [0, 1]])
sigma_x = np.array([[0, 1],
                    [1, 0]])
Id = np.identity(2)

def construct_H3(n, c):
    '''Parameters:
    n: Number of Qubits.
    c: Coupling Coefficient -- not in use, just here to match
    inputs of the other 2 Hamiltonians.
    
    Returns: C-NOT Hamiltonian.''

    H_tot = np.zeros((2**(n+1), 2**(n+1)))
term1 = Id
term2 = sigma_x

    # i corresponds to term number
    for i in range(1, n+1):
        # last term
        if i == n:
            # for term1 must be state0 x term1
            term1 = np.kron(state0, term1)
            # for term2 must be state1 x term2
            term2 = np.kron(state1, term2)
        # for n qubits
        else:
            # Identity matrix x term1 where term starts off as
            # Identity Matrix
            term1 = np.kron(Id, term1)
            # Identity matrix x term2 where term starts off as
            # sigma_x
            term2 = np.kron(Id, term2)

    # final H is the sum of these two terms
    H_tot += term1 + term2

    print("H3: \n", H_tot)
    return(H_tot)

###########################################################################
### Constructing Hamiltonian 4
### ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
### The main function of this module is to construct the
### Heisenberg Hamiltonian
### -- takes in three variables:
###    ----> n: number of coupled qubits
###    ----> c: strength of coupling
```python
### -- outputs H3
###
# Creating the needed Matrices
sigma_x = np.array([[0, 1],
                    [1, 0]])
sigma_y = np.array([[0, 0-1j],
                    [0+1j, 0]])
sigma_z = np.array([[1, 0],
                    [0, -1]])
Id = np.identity(2)

def construct_H4(n, J):
    '''Parameters:
    n: Number of Qubits.
    J: Coupling Coefficient -- not in use, just here to match
    inputs of other 2 hamiltonians.

    Returns: Heisenberg Hamiltonian.'''

    H_tot = np.zeros((2**(n+1), 2**(n+1)), dtype = complex)
terms = [sigma_x, sigma_y, sigma_z]

    # Does this for every subterm
    for i in range(0, 3):
        # First making the list of matrices to be tensored
        kronlist = [terms[i], terms[i]]
        for j in range (1, n):
            kronlist.append(Id)

        # [term, term] // n = 1 ---> run 1 times
        # [term, term, id] // n = 2 ---> run 3 times
        # [term, term, id, id] // n = 3 ---> run 4 times

        # Runs number of total qubits times...
        if n>1:
            for i in range(0, n+1):
                term = kronlist[0]
                # Runs number of env qubits times
```
for k in range(0, n):
    term = np.kron(term, kronlist[k+1])
    H_tot += -J*term
    # Changing order of list
    kronlist.insert(0, (kronlist.pop(n)))
else:
    term = kronlist[0]
    # Runs number of env qubits times
    for k in range(0, n):
        term = np.kron(term, kronlist[k+1])
        H_tot += -J*term

print('H4: ', H_tot)
return H_tot

# Creating Permutation Matrices
### ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
### The main function of this module is to create a permutation
### matrix that will switch the role of "system"
### to each environmental qubit
### -- takes in two variables:
### ----> n: number of total qubits
### ----> slot: number of environmental qubit that you want to
### switch with "system"
### -- outputs the appropriate permutation matrix
#

import numpy as np

def decimalToBinary(num, n):
    '''This function returns a binary number with the appropriate
    amount of leading 0s.
    Parameters:
        num: number to be converted to binary.
        n: expected length of the binary number.
    Returns: Single binary number.'''
    bin = "{:b}".format(int(num))
    if len(bin) == n:
        return "{:0}b".format(int(num))
    else:
```python
return str(bin).zfill(n)

def createperm(n, slot):
    '''This function returns a permutation matrix that switches
    slots in a tensor product.
    Parameters:
    n: number of total qubits.
    slot: slot to be switched with slot A.
    
    Returns: One permutation matrix.''

    slot = slot - 1
    perm = np.zeros((2**n, 2**n))

    # Creating lists of binary numbers to represent states of
    # the qubits
    binlist = []
    for i in range(2**n):
        binlist.append(str(decimalToBinary(i, n)))

    # Iterating through lists to see which are the same
    for i in range(len(binlist)):
        if binlist[i][0] == binlist[i][slot]:
            perm[i][i] = 1
        else:
            numsearch = ''
            for j in range(len(binlist[i])):
                if j == 0:
                    numsearch += binlist[i][slot]
                elif j == slot:
                    numsearch += binlist[i][0]
                else:
                    numsearch += binlist[i][j]
            k = 0
            while binlist[k] != numsearch:
                k += 1
            perm[k][i] = 1

    return(perm)
```

### Creating Initial States

The main function of this module is to create the initial states of the env and the sys -- takes in two variables:

- n: number of coupled qubits
- -- outputs collective initial state

```python
import numpy as np

def init_states(n):
    '''Parameters:
    n: Number of environmental Qubits.

    Returns: Initial state to be evolved forward in time.'''

    # Initial state for env will be a maximally mixed state of
    # the environment, conditions: has to be positive Hermitian
    # matrix with trace = 1
    env_i = (1/(2**n))*np.identity(2**n)

    # Creating list of the composite initial states
    initial_states = []
    for i in range(2):
        sys_i = np.zeros((2, 2), int)
        sys_i[i][i] = 1
        state = np.kron(sys_i, env_i)
        initial_states.append(state)

    return initial_states
```

### Generalized Gell-Mann Matrices

The main function of this module is to construct all GGMM matrices for some dimension D

```python
import numpy as np
import math
import itertools
```
### Old Code that computes GGMMs using equations in AppendixA ###

# Written by Louisa Cornelis

def symmetrical_gmm(j, k, d):
    
    '''Constructs symmetrical ggm matrices.
    Parameters:
    j, k: Indices to be iterated over.
    d: Dimension.
    
    Returns: One symmetrical ggm.''
    
    A = np.zeros((d, d), int)
    A[k][j] = 1
    A[j][k] = 1
    return A

def antisymmetrical_gmm(j, k, d):
    
    '''Constructs antisymmetrical ggm matrices.
    Parameters:
    j, k: Indices to be iterated over.
    d: Dimension.
    
    Returns: One antisymmetrical ggm.''
    
    A = np.zeros((d, d), complex)
    A[j][k] = -1j
    A[k][j] = 1j
    return A

def construct_ggmm(lambdas, d):
    
    '''Constructs full set of symmetrical and antisymmetrical ggmms.
    Parameters:
    lambdas: List of ggmms to append to.
    d: Dimension.
    
    Returns: Antisymmetrical and symmetrical ggmms.''
    
    for k in range(1, d):
        for j in range(k):
            lambdas.append(symmetrical_gmm(j, k, d))
            lambdas.append(antisymmetrical_gmm(j, k, d))
```python
return lambdas

def diagonal_helper(l,d):
    '''Helper function to construct diagonal ggmms.'''
    A = np.zeros((d,d),int)
    for j in range(l+1):
        A[j][j]=1
    return A

def diagonal_ggmm(lambdas,d):
    '''Function to construct diagonal ggmms.
    Parameters:
    lambdas: List of ggmms to append to.
    d: Dimension.

    Returns: List of ggmms with diagonal ggmms appended.'''
    E_1 = np.zeros((d,d),int)
    for l in range(0, d-1):
        coeff = math.sqrt(2/((l+1)*(l+2)))
        E_1[l+1][l+1] = 1
        lambdas.append(coeff*((-(l+1)*E_1)+diagonal_helper(l,d)))
    E_1 = np.zeros((d,d),int)
    return lambdas

def construct_ggmm_sub(d):
    '''Constructs ggmms for a subsystem of dimension d.
    Parameters:
    d: Dimension.

    Returns: List of ggmms of dimension d.'''
    lambdas = []
    construct_ggmm(lambdas,d)
    diagonal_ggmm(lambdas,d)
    return lambdas

def tensor_Ib(matrix, ident_b):
    '''Tensors identity matrix of dimension of subsystem b with matrix.'''
    return np.kron(matrix, ident_b)
```
def tensor_Ia(matrix, ident_a):
    '''Tensors matrix with identity matrix of dimension of subsystem a.'''
    return np.kron(ident_a, matrix)

def construct_all_ggmm(d_a, d_b):
    '''Constructs ggmm for joint system.
    Parameters:
    d_a: Dimension of subsystem a.
    d_b: Dimension of subsystem b.
    Returns: Complete list of ggmm of dimensions d_a*d_b.''

    total = []
a_ggmm = construct_ggmm_sub(d_a)
b_ggmm = construct_ggmm_sub(d_b)
ident_b = np.identity(d_b)
ident_a = np.identity(d_a)
total.extend([tensor_Ib(x, ident_b) for x in a_ggmm])
total.extend([tensor_Ia(x, ident_a) for x in b_ggmm])
    for a in a_ggmm:
        for b in b_ggmm:
            total.append(np.kron(a, b))
    return total

### New Code that computes GGMMs using Pauli Sigma Matrices ###
# Allows us to keep track of couplings

def tensor_func(n, tensorder):
    ''' Function that computes the tensor product of matrices for n qubits
    Parameters:
    n: number of environmental qubits
tensorder: list of matrices to tensor
    Returns: GGMMs of the correct dimension and interaction'''

    term = tensorder[0]
    for i in range(0, n):
        term = np.kron(term, tensorder[i + 1])
    return term

def construct_all_ggmm_nm(d, envqus):
'New method for constructing ggmms for joint system while also keeping track of the GGMMs that belong to a certain system.

Parameters:

d: Dimension of qubits
envqus: Number of environmental qubits

Returns: Complete list of ggmms of dimensions d**total qubits.''

totqus = envqus + 1

# List of sigma_x, sigma_y, sigma_z
sys_ggmm = construct_ggmm_sub(d)

# 2x2 Identity Matrix
ident = np.identity(d)
alpha = [ident, sys_ggmm[0], sys_ggmm[1], sys_ggmm[2]]

# First creating list of lists to be tensored
# Each slot can vary between 0..3
x = [0, 1, 2, 3]
# Applying a permutation with repetition
numlist = [p for p in itertools.product(x, repeat = totqus)]
# Getting rid of the first element because it is not traceable
numlist.pop(0)

# Tesnoring the lists to construct each GGMM
allGGMMs = []
tenslist = []
check = []
for i in range(len(numlist)):
    for j in range(envqus+1):
        # Adding in the corresponding alpha matrix to the # tensor list
tenslist.append(alpha[numlist[i][j]])
        check.append(numlist[i][j])
        # Adding in corresponding GGMM into the total list
allGGMMs.append(tensor_func(envqus, tenslist))
tenslist = []
check = []

return numlist, allGGMMs

# ========================= SYS GGMMS ===========================
# Method to only construct GGMMs of smaller subset using equations
# From AppendixA

def construct_sys_ggmm(lambdas, d):
    '''Constructs full set of symmetrical and antisymmetrical ggmmms for the system.
    Parameters:
    lambdas: List of ggmmms to append to.
    d: Dimension.
    
    Returns: Antisymmetrical and symmetrical ggmmms.'''

    for k in range(1, d):
        for j in range(k):
            lambdas.append(symmetrical_gmm(j, k, d))
            lambdas.append(antisymmetrical_gmm(j, k, d))
    return lambdas

def construct_sys_ggmm_sub(d):
    '''Constructs ggmmms for a subsystem of dimension d.
    Parameters:
    d: Dimension.
    
    Returns: List of ggmmms of dimension d.'''

    lambdas = []
    construct_sys_ggmm(lambdas, d)
    diagonal_ggmm(lambdas, d)
    return lambdas

def construct_allsys_ggmm(d_a, d_b):
    '''Constructs ggmmms for system.
    Parameters:
    d_a: Dimension of subsystem a.
    d_b: Dimension of subsystem b.
    
    Returns: Complete list of ggmmms of dimensions d_a*d_b.'''

    total = []
    a_ggmm = construct_sys_ggmm_sub(d_a)
    ident_b = np.identity(d_b)
    # Additional code
total.extend([tensor_Ib(x,ident_b) for x in a_ggmm])

return total

def construct_unitary(thetas, GGMMs):
    '''Constructs unitary matrix from thetas.
    Parameters:
    thetas: Theta coefficients.

    Returns: Unitary matrix.''

    param_GGMM = []
    for i in range(len(thetas)):
        param_GGMM.append(thetas[i]*GGMMs[i])
    #Sum all scaled matrices then exponentiate.
    scrambler = np.array(expm(1j*sum(param_GGMM)), dtype=complex)

    return scrambler

def reduce_DM_A(matrix, d_sys, d_env):
    '''Parameters:
matrix: Matrix to take trace of.

d_sys: Dimensions of system.

d_env: Dimensions of environment.

Returns: Reduced density matrix of sys.''

```python
dm_vals = []
val = 0

reduced_density = np.zeros((d_sys, d_sys), dtype=complex)
# iteration down matrix
for row in range(0, d_env*2, d_env):
    # iteration across matrix
    for col in range(0, d_env*2, d_env):
        # iteration that sums diagonals
        for k in range(d_env):
            val += matrix[row+k][col+k]
        dm_vals.append(val)
    val = 0
reduced_density = [dm_vals[i:i+d_sys] for i in range(0, len(dm_vals), d_sys)]

# Converting the list into a workable array
red_mat = np.zeros((d_sys, d_sys), dtype = complex)
for i in range(0, d_sys):
    for j in range(0, d_sys):
        red_mat[i][j] = reduced_density[i][j]

return red_mat
```

# Time Evolution

### The main function of this module is to time evolve a reduced density matrix

```python
from scipy.linalg import expm
from numpy.linalg import norm

# CHARACTERISTIC TIME
```
def characteristic_time(Hamiltonian):
    '''Calculates the characteristic time for a given Hamiltonian.
    Parameters:
    Hamiltonian: The hamiltonian matrix.
    Returns: The characteristic time.'''
    # Time inversely related to energy
    return 1/(norm(Hamiltonian, 2))

# ===================== TIME EVOLUTION =====================

def time_evolution(initial_state, H, t):
    '''Time evolution on density matrix.
    Parameters:
    initial state: State to evolve.
    H: Hamiltonian to use in evolution.
    t: Time to evolve to.
    Returns: Evolved state.'''
    return (expm(-(1j)*H*t).dot(initial_state)).dot(expm((1j)*H*t))

# >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
# # Entropy
# # ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# # The main function of this module is to calculate the purity
# # entropy of a given density matrix
# # >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

import numpy as np
from numpy.linalg import matrix_power

def purity_entropy(rho):
    '''Finds the purity entropy given a density operator.
    Parameters:
    rho: density operator.
    Returns: Purity entropy.'''
    return 1-np.matrix.trace(matrix_power(rho, 2))
# Machine Learning Functions

### The main function of this module is to create the cost function and the gradient descent algorithm

###

```python
import numpy as np
from scipy.linalg import expm
from numpy import linalg as LA
import mod_initstates as init
import mod_timeevol as te
import mod_reduce as red
import mod_entropy as ent

# COST FUNCTIONS

# Creating CF1 --- USED IN SYSTEM GRADIENT DESCENT
def func1(GGMM, scramham, n, arb_thetas, init_states, time):
    '''Parameters:
    GGMM: List of Generalized Gell-Mann Matrices.
    scramham: Scrambled Hamiltonian.
    n: Number of environmental qubits.
    arb_thetas: List of theta values that parameterize the Unitary Matrix.
    init_states: State to evolve forward.
    time: Characteristic time

    Returns: Score.''

    # Specifying dimensions
    dsys = 2
denv = (2)**n
D = ((dsys*denv)**2)-1

    # Multiplying arbitrary thetas to the GGMMs
    param_GGMM = []
    for i in range(0, len(arb_thetas)):
        param_GGMM.append(arb_thetas[i]*GGMM[i])

    # Constructing the trial unitary matrix with the GGMMs
    unitary_trial = np.array(expm(1j*sum(param_GGMM)), dtype =
```
scores = []
# Code for testing all scrambled Hamiltonians
# for i in range(0, len(scramham)):
#     H_total = (unitary_trial.dot(scramham[i])).dot(np.conjugate(np.transpose(unitary_trial)))
#     for i in range(0, dsys):
#         evolved_state = te.time_evolution(init_states[i], H_total, time)
#         scores.append(ent.purity_entropy(red.reduce_DM_A(evolved_state, dsys, denv)).real)

# Code for only testing one hamiltonian
H_total = (unitary_trial.dot(scramham)).dot(np.conjugate(np.transpose(unitary_trial)))
for i in range(0, dsys):
    evolved_state = te.time_evolution(init_states[i], H_total, time)
    scores.append(ent.purity_entropy(red.reduce_DM_A(evolved_state, dsys, denv)).real)

# Return average score of both starting states
return sum(scores)/len(scores)

# Creating CF2 --- USED IN CURRENT GRADIENT DESCENT
def func2(GGMM, scramhams, arb_thetas, sigma, env, n, time):
    '''Parameters:
    GGMM: List of Generalized Gell-Mann Matrices.
    scramham: Scrambled Hamiltonian.
    n: Number of environmental qubits.
    arb_thetas: List of theta values that parameterize the Unitary Matrix.
    init_states: State to evolve forward.
    time: Characteristic time.
    
    Returns: Score.''

    # Specifiying dimensions
dsyz = 2
denv = (2)**n
D = ((dsyz*denv)**2)-1
# Multiplying arbitrary thetas to the GGMMs
param_GGMM = []
for i in range(0, len(arb_thetas)):
    param_GGMM.append(arb_thetas[i]*GGMM[i])

# Constructing the trial unitary matrix with the GGMMs
unitary_trial = np.array(expm(1j*sum(param_GGMM)), dtype = complex)

totscores = []
# For each permutated matrix...
for j in range(0, n+1):
    H_total = (unitary_trial.dot(scramhams[j])).dot(np.conjugate(np.transpose(unitary_trial)))
    # Computing Q alphas...
    Q_alphas = []
    for k in range(0, 3): # For alpha = 1, 2, 3
        state = np.kron(sigma[k], env)
        Q_alphas.append(te.time_evolution(state , H_total , time ))
    # Computing Matrix elements
    M = np.zeros((3,3), dtype = complex)
    for row in range(0,3):
        for col in range(0,3):
            a = np.array(red.reduce_DM_A(Q_alphas[row], 2, 2**n))
            b = np.array(red.reduce_DM_A(Q_alphas[col], 2, 2**n))
            M[row][col] = np.trace(a.dot(b))
    # Computing max eigenvalue and corresponding eigenvector
    lam, V = LA.eigh(M)
    maxlam = lam[-1]
    # Adding important info into lists
    totscores.append(1/2 -1/4*( maxlam))
# Computing final score
finalscore = sum(totscores)/len(totscores)

return finalscore

def func2IO(GGMM , scramhams , arb_thetas , sigma , env , n, time):
    ''' Same as func2, but saves n_vecs and other important info
Parameters:
    GGMM: List of Generalized Gell-Mann Matrices.
scramham: Scrambled Hamiltonian.
n: Number of environmental qubits.
arb_thetas: List of theta values that parameterize the Unitary Matrix.
init_states: State to evolve forward.
time: Characteristic time.

Returns: All scores and n vectors.''

# Specifying dimensions
dsys = 2
denv = (2)**n
D = ((dsys*denv)**2)-1

# Multiplying arbitrary thetas to the GGMMs
param_GGMM = []
for i in range(0, len(arb_thetas)):
    param_GGMM.append(arb_thetas[i]*GGMM[i])

# Constructing the trial unitary matrix with the GGMMs
unitary_trial = np.array(expm(1j*sum(param_GGMM)), dtype = complex)

totscores = []
n_vecs = []
# For each permutated matrix...
for j in range(0, n+1):
    H_total = (unitary_trial.dot(scramhams[j])).dot(np.conjugate(np.transpose(unitary_trial)))
    # Computing Q alphas...
    Q_alphas = []
    for k in range(0, 3): # For alpha = 1, 2, 3
        state = np.kron(sigma[k], env)
        Q_alphas.append(te.time_evolution(state, H_total, time))
    # Computing Matrix elements
    M = np.zeros((3,3), dtype = complex)
    for row in range(0,3):
        for col in range(0,3):
            a = np.array(red.reduce_DM_A(Q_alphas[row], 2, 2**n))
            b = np.array(red.reduce_DM_A(Q_alphas[col], 2, 2**n))
\( M[\text{row}][\text{col}] = \text{np.trace}(a \cdot b) \)

# Computing max eigenvalue and corresponding eigenvector
lam, V = LA.eigh(M)
maxlam = lam[-1]
n_vector = V[-1]
# Adding important info into lists
totscores.append(1/2 - 1/4 * maxlam)
n_vecs.append(n_vector)
# Computing final score
finalscore = sum(totscores) / len(totscores)

return totscores, n_vecs, finalscore

# ============== GRADIENT DESCENT ===============

def gradient_descent(gradient, start, learn_rate, n_iter, cost_func, adjust, adjLR):
    '''Parameters:
    gradient: Python function that takes a vector and returns the gradient of the function we want to minimize.
    start: Starting vector.
    learn_rate: Learning rate that controls magnitude of vector update
    n_iter: Number of iterations.

    Returns: Minimizing list of vectors.'''

    theta_trial = start
    cost_history = np.zeros(n_iter)

    for i in range(n_iter):
        # Creating the vector that is equivalent to the "step"
        diff = -learn_rate * gradient(theta_trial)
        # Adding the step to the trial parameters
        theta_trial += diff
        # Computing the score of the new theta_trial
        score_tg = cost_func(theta_trial)
        # Keeping track of how the score changes
        cost_history[i] = score_tg

        # A safety measure that lowers the learning rate
        if i == adjust:
            learn_rate = adjLR
print('Theta final guess score: ', cost_history[-1])
return theta_trial, cost_history

# System Gradient Descent
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# The main function of this module is to run gradient descent
# using only the smaller subset of thetas that allows the
# algorithm to minimize up and down a TPS line
#
# Importing modules
import math
import numpy as np
import numdifftools as nd
import matplotlib.pyplot as plt
import mod_score as sc
import mod_createperm as cp
import mod_initstates as init
import mod_costfunc as cf
import mod_timeevol as te

def runsgd(trial_num, iterations, learning_rate, hamilt_num, numq, GGMMs, H_total, totGGMMs, ggmm_labels, adj_factor, new_lrate):
    # Opening files...
    output_file1 = open(str(trial_num) + '_systrialAVGN_H' + str(hamilt_num) + '_qu' + str(numq+1) + '.txt', 'w')
    output_file2 = open('systrialAVGNscores_H' + str(hamilt_num) + '_qu' + str(numq+1) + '.txt', 'a')
    output_file4 = open('systrialSYS1cores_H' + str(hamilt_num) + '_qu' + str(numq+1) + '.txt', 'a')
    output_file1.write('GRADIENT DESCENT TRIAL ' + str(trial_num) + ' FOR HAMILTONIAN ' + str(hamilt_num) + '
')
    output_file2.write('Running ' + str(iterations) + ' iterations with a learning rate of ' + str(learning_rate) + ' for ' + str(numq+1) + ' total qubits
')
    output_file1.write('*** Permutation matrices are being implemented!
')
    # Scrambled Hamiltonian
scram_thetas = []
# Change only this to sysGGMMs for native TPS scrambling
for i in range(len(totGGMMs)):
    scram_thetas.append(np.random.uniform(-math.pi/4, math.pi/4))
    # Use 0's to stay in native TPS
    # scram_thetas.append(0)
print('Scrambled theta key', scram_thetas)
unitary = sc.construct_unitary(scram_thetas, totGGMMs)
H_scram = (unitary.dot(H_total)).dot(np.conjugate(np.transpose(unitary)))
print(H_scram)
char_time = te.characteristic_time(H_scram)

# Writing scrambled info to file
output_file1.write("\n Theta Key: 
")
np.savetxt(output_file1, scram_thetas)

# Implementing Permutation Matrices
H_scrams = []
H_scrams.append(H_scram)
for i in range(1, numq+1):
    perm = cp.createperm(numq+1, i+1)
    H_scrams.append(perm.dot(H_scram).dot(np.conjugate(np.transpose(perm))))

# Creating initial states -- list of [1 0][0 0] and [0 0][0 1] tensored states
initial_states = init.init_states(numq)

# New set of random thetas for starting iteration
theta0 = [np.random.uniform(-math.pi/4, math.pi/4), np.random.uniform(-math.pi/4, math.pi/4), np.random.uniform(-math.pi/4, math.pi/4)]
print("Starting theta: ", theta0)

# Writing starting info to file
output_file1.write("\n Theta Starting: 
")
np.savetxt(output_file1, theta0)

##### Implementing Machine Learning Alg #####
ind_list = []
totscores = []
for j in range(1, numq + 1):
    
def cost_func(arb_theta):
        score = cf.func1(GGMMs, H_scrams[j-1], numq, arb_theta, initial_states, char_time)
        return score

### QUICK CHECK AT THETA START ###
startscore = cost_func(theta0)
print("Check at starting theta :", startscore)
output_file1.write("\n\n## INFO FOR SYS AS QUBIT \n")
output_file1.write("\n Theta Start Score: " + str(j) + " \n")
output_file1.write("\n Theta Start Score: " + str(startscore) + "\n")

### Code to graph a slice of the cost function -- 3D graph
# x = np.linspace(-math.pi, math.pi, 20)
# y = np.linspace(-math.pi, math.pi, 20)
# coords = []
# vals = []
# for a in range(len(x)):
#     for b in range(len(y)):
#         coords.append([x[a], y[b], 0])
#         val = cost_func([x[a], y[b], 0])
#         vals.append(val)
# # Reassigning xyz-vals according to coords
# X = []
# Y = []
# Z = []
# for i in range(len(coords)):
#     X.append(coords[i][0])
#     Y.append(coords[i][1])
#     Z.append(vals[i])
# fig = plt.figure()
# ax = plt.axes(projection='3d')
# ax.plot_trisurf(X, Y, Z, cmap='viridis', edgecolor='green')
# ax.set_xlabel('theta1')
# ax.set_ylabel('theta2')
# ax.set_zlabel('score')
# ax.set_title('Hamiltonian %d % (hamilt_num) + for %d')
74

%(numq) + ' Env Qubits, Scrambled Hamiltonian %d' %(ii))
    # import pickle
    # plt.savefig(str(trial_num) + '_systrialAVGN_H' + str(hamilt_num) + '_qu' + str(numq+1) + '_ham' + str(ii) + '.png')
    # pickle.dump(fig, open(str(trial_num) + '_systrialAVGN_H' + str(hamilt_num) + '_qu' + str(numq+1) + '_ham' + str(ii) + '.pickle', 'wb'))
    # plt.show()
    # plt.close(1)

    # Opening file that documents cost history
    output_file3 = open(str(trial_num) + '_systrialAVGN_CH_H' + str(hamilt_num) + '_qu' + str(numq+1) + '_ham' + str(j) + '.txt', 'w')

    # Starting Gradient Descent
    grad = nd.Gradient(cost_func)
    # Method to keep running until we've reached a minimum
    slope = -100
    tot_history = []
    while slope < -1E-5:
        print("LR and its: ", learning_rate, iterations)
        theta_qc, cost_history = cf.gradient_descent(grad, theta0, learning_rate, iterations, cost_func, adj_factor, new_lrate)

        # Writing cost_history info into complete history list
        for i in range(len(cost_history)):
            tot_history.append(cost_history[i])

        # Sending last 10 points to be linearly fit
        # -- if last 10 points settle, we have found a minimum
        x = list(range(0, 8))
        y = tot_history[-8:]
        fit = np.polynomial.polynomial.polyfit(x, y, 1)
        slope = fit[1]
        print("Slope of last 10 points: ", slope)

        # Adjusting parameters if we haven't reached the minimum
        theta0 = theta_qc
        # 5 for 2 qubits // 10 for 3 qubits
        iterations = 5
# 0.3 for 2 qubits // 0.6 for 3+ qubits
learning_rate = 0.3
adj_factor = 10
new_lrate = 0.3

# Adding starting score to start of cost history list
tot_history = np.insert(tot_history, 0, startscore)

# Plotting the cost history against iterations to
# verify that we found the minimum
fig, ax = plt.subplots()
ax.set_ylabel('Score')
ax.set_xlabel('Iterations')
ax.set_title('Iterations vs Score with Rate of %d %d' %
learning_rate + ' for Scrambled Hamiltonian %d %d')
ax.set_title('Cost History for Scrambled Hamiltonian 2, 2 Qubits')
_=ax.plot(range(len(tot_history)), tot_history, 'b.' )
plt.show()
plt.close()

# Writing cost history info to file
for i in range(len(cost_history)):
    output_file3.write(str([i, cost_history[i]]) + '
')
output_file3.close()

### QUICK CHECK AT THETA GUESS ###
endscore = cost_func(theta_qc)
print("Check at theta guess: ", endscore)
totscores.append(endscore)

# Writing theta guess info to file
output_file1.write("\n Theta Guess: \n")
np.savetxt(output_file1, theta_qc)
output_file1.write("\n Theta Guess Score: " + str(endscore
 ) + "\n")

# Writing final averaged info to file
thetafinal = sum(totscores)/len(totscores)
output_file1.write("\n\n ## Theta Final Guess Score: " + str(
thetafinal) + " ## \n")
output_file2.write(str([trial_num, thetafinal]) + \n')

# Closing files
output_file1.close()  
output_file2.close()  

return scram_thetas

##################################################################
### Run Gradient Descnet Algorithm
### ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
### The main function of this module is to run gradient descent
### using all theta parameters. This version of the gradient
descent algorithm turns the algorithm into a function that
can be called in runsysfile.py
### -- intermediary step of code that allows things to run
### smoother
###
# Importing modules ###
import math
import numpy as np
import numdifftools as nd
import matplotlib.pyplot as plt
import mod_score as sc
import mod_createperm as cp
import mod_initstates as init
import mod_costfunc as cf
import mod_timeevol as te
from scipy.linalg import expm

def runsgd(trial_num, iterations, learning_rate, hamilt_num, numq, 
GGMMs, H_total, totGGMMs, ggmm_labels, adj_factor, newLR):

    # Opening files...
    output_file1 = open(str(trial_num) + '_systrialAVGN_H' + str(hamilt_num) + '_qu' + str(numq+1) + '.txt', 'w')
    output_file2 = open('systrialAVGNscores_H' + str(hamilt_num) + '_qu' + str(numq+1) + '.txt', 'a')
    # output_file4 = open('systrialAVGNsYS1scores_H' + str(hamilt_num) + '_qu' + str(numq+1) + '.txt', 'a')
    # output_file4 = open('systrialAVGNsYS1scores_H' + str(hamilt_num) + '_qu' + str(numq+1) + '.txt', 'a')
    output_file1.write("GRADIENT DESCENT TRIAL " + str(trial_num) + "+ FOR HAMILTONIAN " + str(hamilt_num) + '\n'
    output_file1.write("Running " + str(iterations) + "+ iterations
with a learning rate of " + str(learning_rate) + " for " +
    
    # Prepare for plotting
    fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 4))
str(numq+1) + ' total qubits 
')
output_file1.write("*** Permutation matrices are being
implemented! 
")

# Scrambled Hamiltonian
scram_thetas = []
for i in range(len(totGGMMs)):
    scram_thetas.append(np.random.uniform(-math.pi/4, math.pi
/4))
unitary = sc.construct_unitary(scram_thetas, totGGMMs)
H_scram = (unitary.dot(H_total)).dot(np.conjugate(np.transpose
(unitary)))
char_time = te.characteristic_time(H_scram)

# Writing scrambled info to file
output_file1.write("\n Theta Key: 
")
np.savetxt(output_file1, scram_thetas)

# Implementing Permutation Matrices
H_scrams = []
H_scrams.append(H_scram)
for i in range(1, numq+1):
    perm = cp.createperm(numq+1, i+1)
    H_scrams.append(perm.dot(H_scram).dot(np.conjugate(np.
transpose(perm))))

# Creating initial states -- list of [1 0][0 0] and [0 0][0 1]
tensored states
# Only needed for original gradient descent method
initial_states = init.init_states(numq)

# New set of random thetas for starting iteration
theta0 = []
for j in range(len(totGGMMs)):
    theta0.append(np.random.uniform(-math.pi/4, math.pi/4))

# Writing starting info to file
output_file1.write("\n Theta Starting: 
")
np.savetxt(output_file1, theta0)

# Setting up initial states for the Matrix Method
# System's initial state - the Pauli Sigma Matrices
sigma_x = np.array([[0, 1],
sigma_y = np.array([[0, 0-1j],
                   [0+1j, 0]]
)
sigma_z = np.array([[1, 0],
                   [0, -1]])
sigmas = [sigma_x, sigma_y, sigma_z]

# Environment's initial state
env_state = (1/(2**numq))*np.identity(2**numq)

##### Implementing Machine Learning Alg #####

# Cost Function - Original Gradient Descent
def cost_func_og(arb_theta):
    score = cf.func1(totGGMMs, H_scrams[1], numq, arb_theta, initial_states, char_time)
    return score

# Cost Function - Matrix Method
def cost_func(arb_theta):
    score = cf.func2(totGGMMs, H_scrams, arb_theta, sigmas, env_state, numq, char_time)
    return score

### QUICK CHECK OF SCORE AT THETA START
startscore = cost_func(theta0)
print("Check at starting theta : ", startscore)
output_file1.write("\n Theta Start Score: " + str(startscore) + "\n")

# Code to graph a slice of the cost function -- 3D graph
# Note -- graphs only work for original gradient descent method since
# matrix method outputs a score for an entire TPS
# x = np.linspace(-math.pi, math.pi, 20)
# y = np.linspace(-math.pi, math.pi, 20)
# coords = []
# vals = []
# for a in range(len(x)):
#     for b in range(len(y)):
#         coords.append([x[a], y[b], 0])
# val = cost_func([x[a], y[b], 0])
# vals.append(val)
# # Reassigning xyz-vals according to coords
# X = []
# Y = []
# Z = []
# for i in range(len(coords)):
#   X.append(coords[i][0])
#   Y.append(coords[i][1])
#   Z.append(vals[i])
# fig = plt.figure()
# ax = plt.axes(projection='3d')
# ax.plot_trisurf(X, Y, Z, cmap = 'viridis', edgecolor = 'green ')
# ax.set_xlabel('theta1 ')
# ax.set_ylabel('theta2 ')
# ax.set_zlabel('score ')
# ax.set_title('Hamiltonian %d' % (hamilt_num) + ' for %d' % (numq) + ' Env Qubits')
# plt.show()

# Opening file that documents cost history
output_file3 = open(str(trial_num) + '_systrialAVGN_CH_H' + str(hamilt_num) + '_qu' + str(numq+1) + '.txt', 'w')

# Starting Gradient Descent
# Method to keep running until we've reached a minimum!!!
slope = -100
# tot_history = []
while slope < -1E-5:
   grad = nd.Gradient(cost_func)
   print("LR and its: ", learning_rate, iterations)
   theta_qc, cost_history = cf.gradient_descent(grad, theta0,
                                               learning_rate, iterations, cost_func, adj_factor, newLR)

   # Writing cost_history info into complete history list
   for i in range(len(cost_history)):
      tot_history.append(cost_history[i])

   # Sending last 10 points to be linearly fit
   # -- if last 10 points settle, we have found a minimum
   x = list(range(0, 10))
   y = tot_history[:-10]
```python
fit = np.polynomial.polynomial.polyfit(x, y, 1)
slope = fit[1]
print("Slope of last 10 points: ", slope)

# Adjusting parameters if we haven't reached the
# minimum
theta0 = theta_qc
iterations = 10
learning_rate = 1.2
adj_factor = 6
newLR = 0.6

# Adding starting score to start of cost history list
tot_history = np.insert(tot_history, 0, startscore)

# Plotting the total cost history against iterations to verify
that we found the minimum
fig, ax = plt.subplots()
ax.set_ylabel('Score')
ax.set_xlabel('Iterations')
ax.set_title('Cost History for Hamiltonian 2, 2 Qubits')
_ = ax.plot(range(len(tot_history)), tot_history, 'b.')
plt.show()
plt.close()

# Writing total cost history info to file
for l in range(len(tot_history)):
    output_file3.write(str([l, tot_history[l]]) + '
')
output_file3.close()

### QUICK CHECK THAT AT THETA GUESS ###
allscores, alln_vecs, endscore = cf.func2IO(totGGMMs, H_scrams,
theta_qc, sigmas, env_state, numq, char_time)
print("Check at theta guess: ", endscore)

# Writing theta guess info to file
output_file1.write("\n Theta Guess: \n")
np.savetxt(output_file1, theta_qc)
output_file1.write("\n ## Theta Final Guess Score: "+str(endscore)+'## \
")
output_file2.write(str([trial_num, endscore]) + '
')
output_file1.write("\n ## PERMUTATION INFO ## \
")
```
# Writing individual N_vector and scoring info to file
for i in range(len(allscores)):
    output_file1.write("\n N Vector for Qubit " + str(i+1) + ": \n")
    output_file1.write(str([alln_vecs[i][0], alln_vecs[i][1],
                              alln_vecs[i][2]]) + "\n")
    output_file1.write("\n Corresponding Score for Qubit " +
                      str(i+1) + ": \n")
    output_file1.write(str(allscores[i]) + 'n')

# Constructing the output H to be checked in LU Checks
Hnew_scrams = []
param_GGMM = []
for i in range(0, len(theta_qc)):
    param_GGMM.append(theta_qc[i]*totGGMMs[i])
unitary_trial = np.array(expm(1j*sum(param_GGMM)), dtype=complex)
for j in range(len(H_scrams)):
    Hnew_scrams.append((unitary_trial.dot(H_scrams[j])).dot(np.conjugate(np.transpose(unitary_trial))))

# Write passed H_scrams to file
for i in range(len(H_scrams)):
    H_scram = H_scrams[i]
    H_els = []
    for row in range(0, 2**(numq+1)):
        for col in range(0, 2**(numq+1)):
            H_els.append(H_scram[row][col])
    output_file1.write("\n Newfound TPS From Gradient Descent :
")
    output_file1.write(str(H_els) + 'n')

# Closing files
output_file1.close()
output_file2.close()
import numpy as np
from ast import literal_eval
import mod_hamilt1 as H1
import mod_hamilt2 as H2
import mod_hamilt3 as H3
import mod_hamilt4 as H4
import mod_createperm as cp

# Pauli Sigma Matrices
sigma_x = np.array([[0, 1], [1, 0]])
sigma_y = np.array([[0, 0-1j], [0+1j, 0]])
sigma_z = np.array([[1, 0], [0, -1]])
sigma = [sigma_x, sigma_y, sigma_z]
Id = np.identity(2)

def getvals(H, numq):
    h_selfs = []
    # First creating the arbitrary tensor order for each qubit
    # For each qubit
    for i in range(0, numq+1):
        kronlist = []
        kron_list = [Id]*(numq+1)
        for j in range(0, 3):
            kron_list[i] = sigma[j]
        # Tensoring the current list
        term = kron_list[0]
        for k in range(0, numq):
            term = np.kron(term, kron_list[k+1])
# Now that we have the proper term, we can input it into the equation
el = np.trace(1/((numq+1)**2)*term.dot(H))
h_selfn.append(el)
# Add the h_selfn vector into the list of h_self vectors
h_self.append(h_selfn)

### Constructing H_ints ###

h_int = []

# h_int master list order:
# 2 qubits: 12
# 3 qubits: 12, 23, 13

# First creating the arbitrary tensor order for the sequences that go down the line
# i.e. 12, 23 ...
for l in range(0, numq):
    kron_list = [Id] * (numq+1)
    # Altering the tensor order according to qubit slot
    # and vector element
    h_intn = []
    h_intM = np.zeros((3, 3), dtype = complex)
    for m in range(0, 3):
        for n in range(0, 3):
            kron_list[l] = sigma[m]
            kron_list[l+1] = sigma[n]
            # Tensoring the current list
            term = kron_list[0]
            # Runs number of env qubits times
            for o in range(0, numq):
                term = np.kron(term, kron_list[o+1])
            # Now that we have the proper term, we can input it into the equation
            el = np.trace(1/((numq+1)**2)*term.dot(H))
            h_intM[m][n] = el
            # Adding matrix into master list
            h_int.append(h_intM)

### Additional matrices for 3 qubits ###
if numq > 1:
    # Constructing 13 by hand
    kron_list = [Id] * (numq+1)
```python
h_intM = np.zeros((3,3), dtype = complex)
for p in range(0,3):
    for q in range(0,3):
        kron_list[0] = sigma[p]
        kron_list[2] = sigma[q]
        # Tensoring the current list
        term = kron_list[0]
        for r in range(0, numq):
            term = np.kron(term, kron_list[r+1])
        # Now that we have the proper term, we can input
        # it into the equation
        el = np.trace(1/((numq+1)**2)*term.dot(H))
        h_intM[m][n] = el
h_int.append(h_intM)

# Constructing the 3x9 matrices
# list order: 1|23, 2|13, 3|21
h_int_3 = []
kron_list = [0]*3
h_intM = np.zeros((3,9), dtype = complex)
# For each Matrix
for v in range(0,3):
    h_intM_els = []
    for s in range(0,3):
        for t in range(0,3):
            for u in range(0,3):
                list_ord = [[s, t, u], [t, u, s], [t, s, u]]
                ord = list_ord[v]
                kron_list[0] = sigma[ord[0]]
                kron_list[1] = sigma[ord[1]]
                kron_list[2] = sigma[ord[2]]
                # Tensoring the current list
                term = kron_list[0]
                for r in range(0, numq):
                    term = np.kron(term, kron_list[r+1])
                # Now that we have the proper term, we can
                # input it into the equation
                el = np.trace(1/((numq+1)**2)*term.dot(H))
                h_intM_els.append(el)
    # Turning each list of elements into 3x9 matrix to be
    # added to h_int
    els = np.array(h_intM_els)
```
h_int_3.append(els.reshape(3,9))

# ==================== TWO QUBIT VALUES ====================

if numq == 1: # For 2 qubits

    # Series 1 checks
    mu = [0, h_self[0], h_int[0].dot(h_self[1]), h_int[0].dot(np.transpose(h_int[0])).dot(h_self[0]), h_int[0].dot(np.transpose(h_int[0])).dot(h_self[0]), h_int[0].dot(np.transpose(h_int[0])).dot(h_self[1]), h_int[0].dot(np.transpose(h_int[0])).dot(h_int[0]).dot(h_self[0]), h_int[0].dot(np.transpose(h_int[0])).dot(h_int[0]).dot(np.transpose(h_int[0])).dot(h_int[0]).dot(h_self[1])]

    # Series 2 checks
    nu = [0, h_self[1], np.transpose(h_int[0]).dot(h_self[0]), np.transpose(h_int[0]).dot(h_int[0]).dot(h_self[1])]

    # Compute 9 values that must be invariant for both H's

    # First 6 values
    checks = []
    for m in range(1,4):
        checks.append(np.inner(mu[m], mu[m]))
        checks.append(np.inner(nu[m], nu[m]))
        checks.append(np.inner(mu[1], mu[m*2]))

    # Last 3 values
    checks.append(np.trace(h_int[0].dot(np.transpose(h_int[0])).dot(h_int[0])))
    checks.append(np.trace(h_int[0].dot(np.transpose(h_int[0])).dot(h_int[0]).dot(np.transpose(h_int[0])))
    checks.append(np.linalg.det(h_int[0]))

    # Rounding to 5th decimal
    rounded_checks = []
    for n in range(len(checks)):
        rounded_checks.append(round(checks[n], 5))

# ==================== THREE QUBIT VALUES ====================
if numq == 2: # For 3 qubits

    # First defining italicized T's
    T1 = h_int_3[0].dot(np.transpose(h_int_3[0]))
    T2 = h_int_3[1].dot(np.transpose(h_int_3[1]))
    T3 = h_int_3[2].dot(np.transpose(h_int_3[2]))
    T23 = np.transpose(h_int_3[0]).dot(h_int_3[0])
    T13 = np.transpose(h_int_3[1]).dot(h_int_3[1])
    T12 = np.transpose(h_int_3[2]).dot(h_int_3[2])

    # Series checks
    S1 = []
    S2 = []
    S3 = []

    # First elements
    for i in range(1,4):
        term1 = T1
        term2 = T2
        term3 = T3
        for j in range(0,i-1):
            term1 = term1.dot(T1)
            term2 = term2.dot(T2)
            term3 = term3.dot(T3)
        if i == 1:
            term1 = term2 = term3 = np.identity(3)
        S1.append(term1.dot(h_self[0]))
        S2.append(term2.dot(h_self[1]))
        S3.append(term3.dot(h_self[2]))

    # Second elements
    for j in range(1,4):
        term1 = T1
        term2 = T2
        term3 = T3
        for j in range(0,j-1):
            term1 = term1.dot(T1)
            term2 = term2.dot(T2)
            term3 = term3.dot(T3)
        if i == 1:
            term1 = term2 = term3 = np.identity(3)
        S1.append(term1.dot(h_int[0]))
        S2.append(term2.dot(np.transpose(h_int[0])))
        S3.append(term3.dot(np.transpose(h_int[2])))
# Third elements
for i in range(1,4):
    term1 = T1
term2 = T2
term3 = T3
    for j in range(0,i-1):
        term1 = term1.dot(T1)
term2 = term2.dot(T2)
term3 = term3.dot(T3)
    if i == 1:
        term1 = term2 = term3 = np.identity(3)
S1.append(term1.dot(h_int[2]))
S2.append(term2.dot(h_int[1]))
S3.append(term3.dot(np.transpose(h_int[1])))

# Fourth elements
for i in range(1,4):
    term1 = T1
term2 = T2
term3 = T3
    for j in range(0,i-1):
        term1 = term1.dot(T1)
term2 = term2.dot(T2)
term3 = term3.dot(T3)
    if i == 1:
        term1 = term2 = term3 = np.identity(3)
S1.append(term1.dot(T1))
S2.append(term2.dot(T2))
S3.append(term3.dot(T3))

# Computing values for the 3 qubit case
from itertools import combinations_with_replacement
# list of i's and j's
comb1 = list(combinations_with_replacement([1, 2], 2))
# list of i's and j's and k's
comb2 = list(combinations_with_replacement([1, 2], 3))

checks = []
for i in range(0, len(comb1)):
    ii = comb1[i][0]
    jj = comb1[i][1]
    checks.append(np.inner(S1[ii], S1[jj]))  # mus
checks.append(np.inner(S2[ii], S2[jj]))  # nus
checks.append(np.inner(S3[ii], S3[jj]))  # omega

# Triple products
for j in range(0, len(comb2)):
    iii = comb2[j][0]
    jjj = comb2[j][1]
    kkk = comb2[k][2]
    # mus
    checks.append(np.inner(S1[iii], np.cross(S1[jjj], S1[kkk])))
    # nus
    checks.append(np.inner(S2[iii], np.cross(S2[jjj], S2[kkk])))
    # omega
    checks.append(np.inner(S3[iii], np.cross(S3[jjj], S3[kkk])))

rounded_checks = []
for n in range(len(checks)):
    rounded_checks.append(round(checks[n], 5))

return checks

def compvals(H_1, H_2, nevs):
    checks_H1 = getvals(H_1, nevs)
    checks_H2 = getvals(H_2, nevs)

    # Printing side by side
    for i in range(len(checks_H1)):
        print("Value ", i, " for H1: ", checks_H1[i], " for H2: ", checks_H2[i])

    # Sorting both the lists
    checks_H1.sort()
    checks_H2.sort()

    # Check if lists are equal
    if checks_H1 == checks_H2:
        print("H is equal to H_native")
    else:
        print("H is not equal to H_native")
### Running Code

The main function of this file is to grab from all main modules and continually run algorithms for n trials -- everything can be adjusted here at the top -- to run all code: run runsysfile

```
import mod_hamilt1 as H1
import mod_hamilt2 as H2
import mod_hamilt3 as H3
import mod_hamilt4 as H4
import mod_GGMM as gg
import run_sys_eig as re
import run_sys_gd as rsg
import run_gd_unfixed as rgu

# Input quick changes...
runnum = 10  # Number of times to run
nqs = 1  # Number of env qubits
hamnum = 2  # Hamiltonian number
cc = 1  # Coupling coefficient
its = 15  # Iterations for gradient descent
lrate = 0.58  # Learning Rate
adj = 20  # Adjustment iteration
newlrate = 0.3  # Adjusted learning rate - half original rate

# H2QU2 -- lrate:0.6 its:15, adj:10, newlrate:0.3
# H2QU3 -- lrate:2.3 its:35, adj:20, newlrate:0.6

### DEFINING VARIABLES ###

# Specifying Dimension
d_sys = 2
d_env = 2**nqs

# Constructing GGMMs
sysGGMMs = gg.construct_allsys_ggmm(d_sys,d_env)
ggmmlabels, allGGMMs = gg.construct_all_ggmm_nm(d_sys, nqs)

# Native Hamiltonian
if hamnum == 1:
```
Htot = H1.construct_H1(nqs, cc)

elif hamnum == 2:
    Htot = H2.construct_H2(nqs, cc)

elif hamnum == 3:
    Htot = H3.construct_H3(nqs, cc)

else:
    Htot = H4.construct_H4(nqs, cc)

trial_num = 1

for i in range(runnum):
    print("############# Trial ", trial_num, " ############")

    # Run for gradient descent method
    print("------------- GRADIENT DESCENT OUTPUT -------------")
    rgu.runsgd(trial_num, its, lrate, hamnum, nqs, sysGGMMs, Htot, allGGMMs, ggmmlabels, adj, newlrate)

    trial_num += 1